Programmierung 2
Studiengang MI / WI

- Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm
- Hochschule Harz
- FB Automatisierung und Informatik
- mwilhelm@hs-harz.de
- http://mwilhelm.hs-harz.de
- Raum 2.202
- Tel. 03943 / 659 338

Inhalt der Vorlesung

Überblick:

- **Objekte und Methoden**
- Swing
- Exception
- I/O-Klassen / Methoden
- Algorithmen (Das Collections-Framework)
- Design Pattern
- JUnit
- Graphentheorie

- Folien basierend auf Vorlesung „Programmierung2“ von Prof. Dr. Drögehorn

Was machen wir hier ?

- Sie **WOLLEN** programmieren lernen !!!
- Objektorientierte Programmierung
  - In der Vorlesung ”Programmierung 2” lernen Sie die entsprechenden in der Sprache Java implementierten Konzepte der objektorientierten Programmierung kennen. Die Tutorien vertiefen die Vorlesungsinhalte und bieten praktische Beispiele.
- Vorgehen
  - Der Kurs erfordert eine aktive Erarbeitung der Lehrinhalte. Dies impliziert zusätzlich ein beträchtliches Selbststudium, inklusive Vor- und Nachbereitung der Vorlesungen an Hand der angegebenen Literatur als auch die eigenständige Durchführung gestellter Programmieraufgaben.
- **ACHTUNG:** Sie lernen Programmierung **NICHT** durchs Zuhören

Literatur

- D. J. Barnes, M. Köllig; Java lernen mit BlueJ, 3. Auflage, 2006, Pearson Sudium
- C. Heinisch, F. Müller, J. Goll; Java als erste Programmiersprache, 4. Auflage, 2006, Teubner
- C. S. Horstmann, G. Cornell; Core Java, Volume 1, 7. Auflage, Prentice Hall
- C. S. Horstmann, G. Cornell; Core Java, Volume 2, 7. Auflage, Prentice Hall
- R. Sedgwick, Algorithms in Java, Parts 1-4, 3. Auflage, Addison Wesley

vertiefende Literatur

- R. Sedgwick, Algorithms in Java, Part 5, 3. Auflage, Addison Wesley
- M. Loy, et. al. Java Swing, O'Reilly, 2nd Edition, 2002
- B. Daum. Java-Entwicklung mit Eclipse 3, dpunkt Verlag, 2004

Einführung / Organisation

- Organisatorisches
  - Programmierung II Mo 09:45 – 13:00 HS C
  - Tutorial Mo 11:30 – 13:00 5.203, 5.206, 5.102
    - 3 Gruppen zu obigem Termin
    - Lösungen einzureichen bis Montag folgender Woche 12.00 Uhr (StudIP)
- Formalia
  - Prüfung: Klausur, 120 min
  - Vorlesungs + Übungszeit: 15(12)x2x1:5h = 45h
- Nach- und Vorbereitung: 105 h (!)

Einführung / Prüfungsleistung

- **Prüfungsleistungen:**
  - Klausur 120 Min.
  - Testat per Übungen
- **Übungen**
  - Drei Gruppen, mit je einem Tutor
    - In jeder Übung wird eine Anwesenheitsliste geführt. In den Übungen werden die gestellten Programmieraufgaben von Studierenden vorgetragen und mit den Tutoren diskutiert. Danach werden von den Tutoren vorbereitete Übungen durchgeführt
  - Abgabe
    - Die in der Vorlesung gestellten Übungsaufgaben sind jeweils spätestens am Mittwoch der folgenden Woche auf StudIP in das entsprechende Verzeichnis hochzuladen. (Abgabe allein oder max. in 2er Grupen: Zip-Datei, Name in der Form mXXXXX_mXXXXX_nn.zip).
  - Testat wird erteilt wenn:
    - Anwesenheit mind. 75% nachgewiesen durch Anwesenheitslisten
    - Die Übungsaufgaben insgesamt mind. zu 75% gelöst wurden

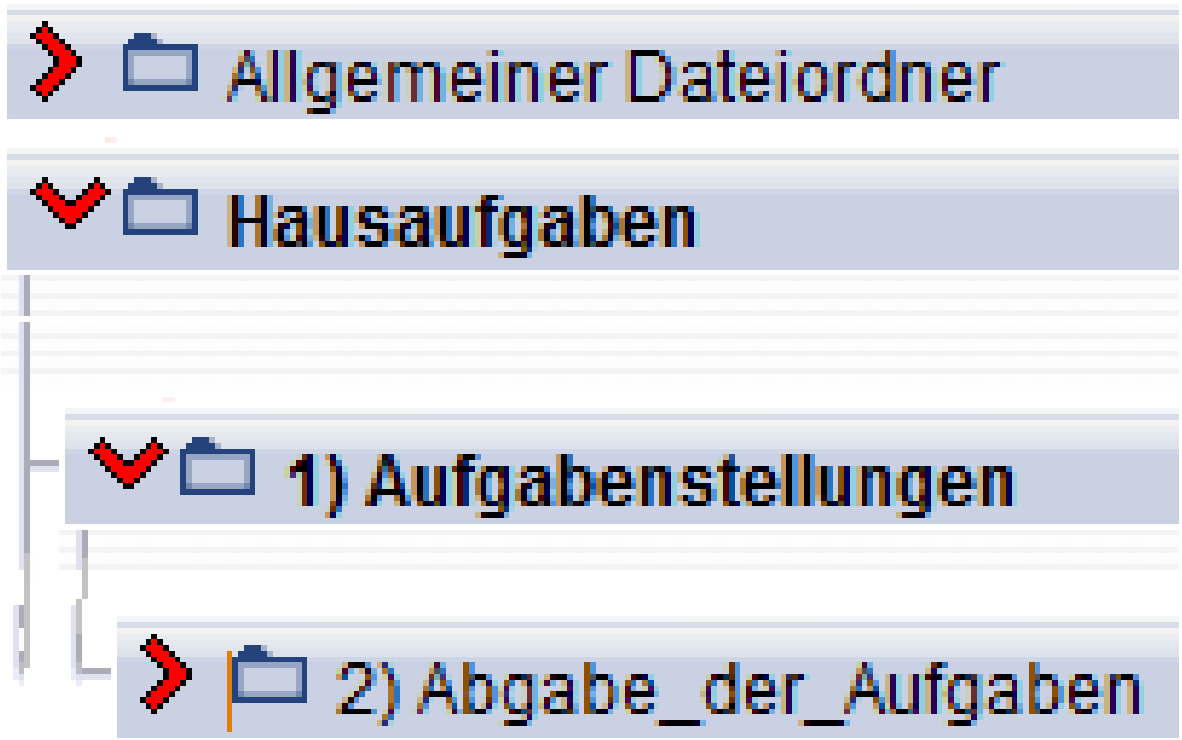
Scripte / Aufgaben / Lösungs-Abgabe/Lerntagebuch

Vorlesung: Programmierung2

Scripte

Aufgaben

Lösung abgeben



Scripte / Aufgaben / Lösungs-Abgabe / Lerntagebuch

Abgaben als Dateien in entsprechende Verzeichnisse in StudIP:

Vorlesung in StudIP:

Programmierung2

Aufgaben bis Montag 08:00 Uhr vor der Vorlesung

(1 Woche nach der Vorlesung),

Ordner: ***Abgabe_Aufgaben***

dann Ordner der Ausgabe auswählen

Lerntagebuch:

Abgabe bis Sonnabend 12:00 Uhr in der Woche der Vorlesung

Hausaufgaben:

pde-Datei/java-Datei gezipped mit **Matrikelnummer**

- Dateiname: 12345_aufgabe01.zip
- Dateiname: 12345__12346_aufgabe01.zip

Die Zip-Datei enthält ein Verzeichnis mit dem Namen der Datei (z.B. XXXXX_01), in diesem Verzeichnis befindet sich der kommentierte Quellcode der Programme.

Skript und Aufgabenstellungen:

- jeweils nach der Vorlesung auf StudIP bzw. meiner Homepage

Kapitel OOP

- Objekte
  - Attribute
  - Methoden
  - Überschreiben (Methoden in mehreren Klassen)
    - **Overriding**
  - Überladen (Methoden einer Klasse)
    - Overloading
- Vererbung
  - Polymorphie
- Interface

Klassen und Objekte

Teil 1: vordefinierte Klassen und Bibliotheken

Was ist ein Objekt?

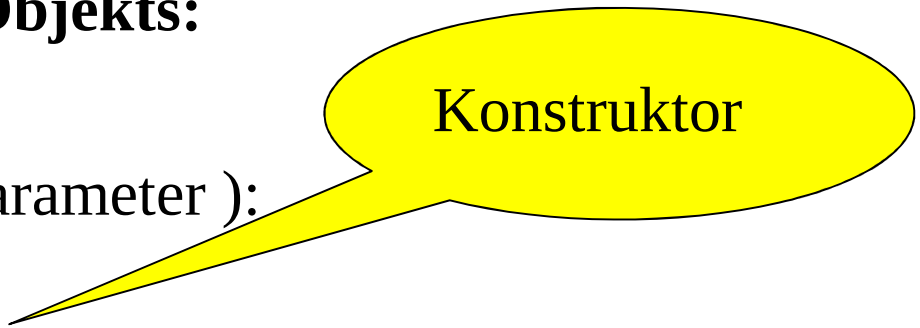
- basiert auf “Hauptwort/Substantiv” (ein Auto, ein Laden,...)
- hat Eigenschaften (Attribute genannt).
- hat Aktionen (Methoden) die ausgeführt werden können (Verben)
- Objekte sind (selbst erstellte) Datentypen.
- Diese Datentypen heißen Klassen.
- Objekte werden immer mit “**call by reference**” übergeben

Erzeugen einer Klasse:

- class name {
  - …
- }

Aufruf der Methoden eines Objekts:

- Erzeugen eines Objekts
- objektname.methode(Parameter):




```
Objekt objektname = new Objekt(<parameterliste>);
objektname.methode()
```

Konstruktor:

- sorgt dafür, dass Objektattributen Anfangswerte zugewiesen werden.
- Hat immer den Klassennamen.

Klassen und Objekte

Klasse:

- Eine Blaupause/Konstruktionsplan für ein Objekt.
- Die allgemeine Idee hinter einem Ding
- Hat Platzhalter für die Details (Attribute)
- Deklariert Funktionen/Methoden, die auf ein Objekt angewandt werden können.

Objekt:

- Ein Objekt ist eine Instanz einer Klasse.
- Das “echte” Ding mit all seinen Eigenschaften
- Seine Methoden können aufgerufen werden
- Jedes Objekt kennt die Klasse, von der es konstruiert wurde.
- Man muss einen Plan haben, bevor man ein Objekt erzeugen kann: Klasse kommt **vor** dem Objekt

Klassen und Objekte

Instanziierung

- Die Erzeugung eines Objekts mittels des in der Klasse beschriebenen Konstruktionsplanes heißt Instanziierung der Klasse.
- Entsprechend handelt es sich bei dem Objekt um eine Instanz der Klasse.

Definition:

```
class CName {
   public CName() {
   }
   public methode1() {
   }
}
```

Benutzung:

```
CName cobj;
cobj = new CName();
cobj.Methode1();
```

Objekte und Attribute

- Objekte innerhalb einer Klasse besitzen identische Attribute, können sich aber in den Attributwerten unterscheiden

- Beispiel
  - Ein Objekt vom Typ Person besitzt ein Attribut des Typs Geschlecht, welches die Werte männlich oder weiblich annehmen kann.

Siehe OOAD

- Prinzip
  - Objektorientierte Programmierung nutzt das Prinzip vom Verstecken der Information (Data Hiding / Kapselung)

- Begründung
  - Die Attribute beschreiben den inneren Zustand eines Objektes. Es ist Vorteilhaft diesen als “Black Box” anzusehen. Die Veränderung des Objektzustandes erfolgt nur über eine (vom Programmierer) wohl definierte Schnittstelle, welche durch Methoden/Operationen implementiert wird.

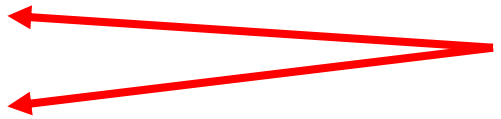
Signatur von Methoden

- Die Signatur einer Methode besteht aus dem Methodennamen und den Datentypen der Parameter in der gegebenen Reihenfolge.
- Der Rückgabewert einer Methode ist nicht Teil der Signatur.
- Methoden in einer Klasse müssen sich durch ihre Signatur unterscheiden.
  - Das heißt, solange die Parameterzahl oder Parametertypen unterschiedlich sind, können Methoden denselben Namen haben.
- Solche Methoden heißen **überladen** (eine Klasse)

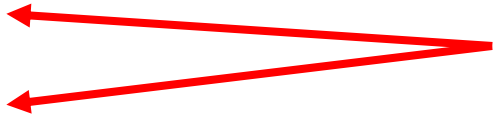
```
class A {
  float methode() { .... }
  float methode(int i) { .... }
  float methode(string s) { .... }
}
```

Welche Methoden haben keine unterschiedliche Signatur?

```
class A {

1  int calc(int i) { ... }
2  float calc(int i) { ... }

3  float calc(float f) { ... }
4  int calc(int i, int j) { ... }
5  int calc(int i, int k) { ... }

6  int calc(int i, float f) { ... }
7  int calc(float f, int i) { ... }

}
```



?

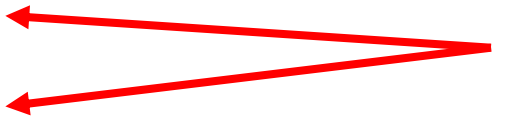


?

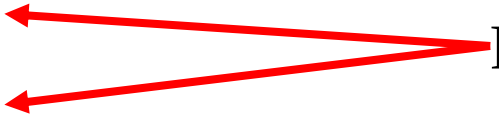
?

Welche Methoden haben keine unterschiedliche Signatur?

```
class A {

1  int calc(int i) { ... }
2  float calc(int i) { ... }

3  float calc(float f) { ... }
4  int calc(int i, int j) { ... }
5  int calc(int i, int k) { ... }

6  int calc(int i, float f) { ... }
7  int calc(float f, int i) { ... }

}
```



Fehler



Fehler

unterschiedlich,
aber schlechter Stil

Konstruktoren

- Die Aufgabe eines **Konstruktors** ist es, die Attribute des Objekts mit (sinnvollen) Werten zu initialisieren.
- Konstruktoren sind Methoden mit dem Namen der Klasse. Sie können überladen werden, das heisst, es kann sie mehrfach geben. Jedoch haben sie keinen Rückgabewert, auch nicht “void”.

```
class A {
  int i;
  String s;
  // default Constructor
  public A() {
    i = 0;
    s = "";
  }
```

```
  // 2. Constructor
  public A(int i, String s){
    this.i = i;
    this.s = s;
  }

} // class A
```

Konstruktoren

```
class A {
  int i;
  String s;
  // default Constructor
 // Aufruf des 2. Const
  public A() {
    this(0,"");
  }

  public A(int i, String s){
   this.i = i;
   this.s = s;
  }
// Aufruf des 2. Const
  public A(int i) {
    this(i,"");
  }

  public A(String s){
   this(0,s);
  }
```

Konstruktoren

```
class Student {
  int mnr;
  String name;
  // default Constructor
  // Aufruf des 2. Construktor
  public Student(int mnr, String name){
   this.mnr = mnr;
   this.name = name;
  }
  // Copy-Konstruktor
  public Student (Student std){
   this.mnr = std.mnr;
   this.name = std.name;
  }
```

Instanzierung und Konstruktoren

- `Student s1 = new Student(12345, "Meier");`
- `Student s2 = new Student(s1); // copy Const.`

  - Das Schlüsselwort this:
  - this steht innerhalb der Methoden einer Klasse für eine Referenz auf die **aktuelle** Instanz des aktuellen Objekts.

```
public Student (Student std){
   this.mnr = std.mnr;
   this.name = std.name;
}
```

Klassenattribute

- Es kommt vor, dass Attribute für alle Instanzen einer Klasse denselben Wert besitzen.
- Dann ist es unnötig in jeder Instanz eine Kopie dieses Attributs zu haben.
- Stattdessen möchte man diesen Wert direkt in der Klassendefinition speichern.
- Dies geschieht mit dem Schlüsselwort **static**.

```
class A {
  static int anzahlInstanzen = 0;
  public A() {
     anzahlInstanzen++;
     ....
   }
}  // class A
```

Klassenmethoden

- Analog kommt vor, dass Methoden sich nicht auf die Attributwerte einer Instanz beziehen.
- Solche Methoden werden unabhängig von Instanzen über den Klassennamen aufgerufen. Sie werden mit dem Schlüsselwort **static** deklariert.
- Häufig dienen diese dazu, Klassen mit „Basisfunktionalität“ auszustatten
- statische Methoden **dürfen nicht** auf dynamische bzw. allgemeine Variablen zugreifen.
- Jede statische Methode ist für sich komplett separat.
- Sie kann aber andere statische Methoden aufrufen.

Statische Methode

Dynamische Methoden:

- Diese Methoden nehmen Bezug auf ein echtes Objekt. Das heißt, dass jede dynamische Methode auf normalen Variablen zugreifen kann. Jedes Objekt hat genau dieselbe Methode, aber jede Methode eines Objektes greift auf unterschiedliche Variablen zu.

```
class Math {
public float quadrat(float x) {
        return x*x;
}
}
```

Aufruf:

```
float x, y;
x=3;
Math m = new Math();
y = m.quadrat(x);
```

Statische Methode

Statische Methoden:

- Eine statische Methode darf nicht auf dynamische Variablen zugreifen. Man benutzt sie häufig, um einfache Berechnung durchzuführen, die keinen Bezug auf ein echtes Objekt nehmen, also unabhängig von den Variablen sind. Es kann aber trotzdem sein, dass man eventuell auch ein, zwei oder drei Objekte einer Klasse als Parameter übergeben kann. In der Klasse „Math“ gibt es die statische Methode „sin“. Dieses ist sinnvoll, da die Funktion Sinus völlig unabhängig agieren kann.
- Ohne diesen Mechanismus müsste man jedes Mal eine Instanz erstellen.
- Statische Methoden werden mittels des Klassennamens aufgerufen.

Statische Methode

Definition:

```
class Math {
  public static float quadrat(float x) {
          return x*x;
  }
}
```

Aufruf:

```
float x, y;
x=3;
y = Math.quadrat(x);  // Klassennamen, kein Objekt
```

Klassenmethoden

```
class Mathe {
static float polynom(float polynomKoeff, float x) {
     // Horner Schema
   float y = polynomKoeff[0];
   for(int i=1; i<polynomKoeff.length; i++) {
     y = y*x + polynomKoeff[i];
   }
   return y;
  }  // polynom
}
```

Aufruf:

```
// y= 3*x*x + 4*x + 2 = ((3*x + 4)*x + 2)
float[] koeff={ 3.0f, 4.0f, 2.0f };
float erg = Mathe.polynom(koeff, 2.0f); // Klassenname
!
```

Objektorientierte Programmierung und das Prinzip vom Verstecken der Information

- Die Attribute einer Instanz beschreiben den **inneren Zustand** des Objektes.
- Es ist vorteilhaft, diesen als “Black Box” anzusehen. Die Veränderung des Objektzustandes erfolgt nur über eine (vom Programmierer) wohl definierte Schnittstelle, welche durch Methoden/Operationen implementiert wird.
- In Java stehen hierfür die Schlüsselworte
  - public,
  - protected
  - private
- zur Verfügung. Sie erlauben den Zugriff von außerhalb einer Klasse auf Attribute und Methoden zu verhindern (private) oder zu erlauben (public).

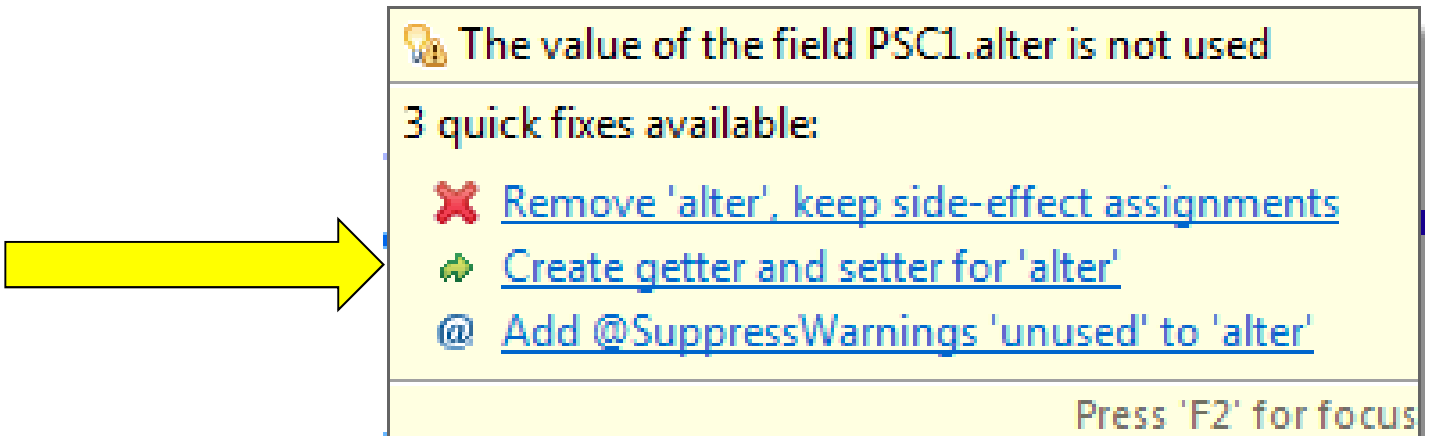
get- und set-Methoden

```
class A {
  private int anzahl;
  ....
  public A() {
    anzahl = 0;
  }
  public void setAnzahl(int n) {
   if (n>0)
     anzahl = n;
   else
     throws Exception.. Error
  }
  public int getAnzahl() {
   return anzahl;
  }
} // class A
```

Automatische Erstellung in Eclipse

get- und set-Methoden

- Methoden zum Lesen und Schreiben von Attributen heißen getter/setter Methoden. Ihre Namen folgen den Konventionen für Funktionen, wobei der erste Namensteil entweder set oder get ist, der zweite Namensteil ist der Name des Attributs.
- In Eclipse kann man das automatisch generieren lassen
  - Das Attribut auf „private“ setzen.
  - Mit der Maus über das Attribut gehen.
  - Eintrag „Create getter and setter for 'alter' “ auswählen.



Die minimale Klasse

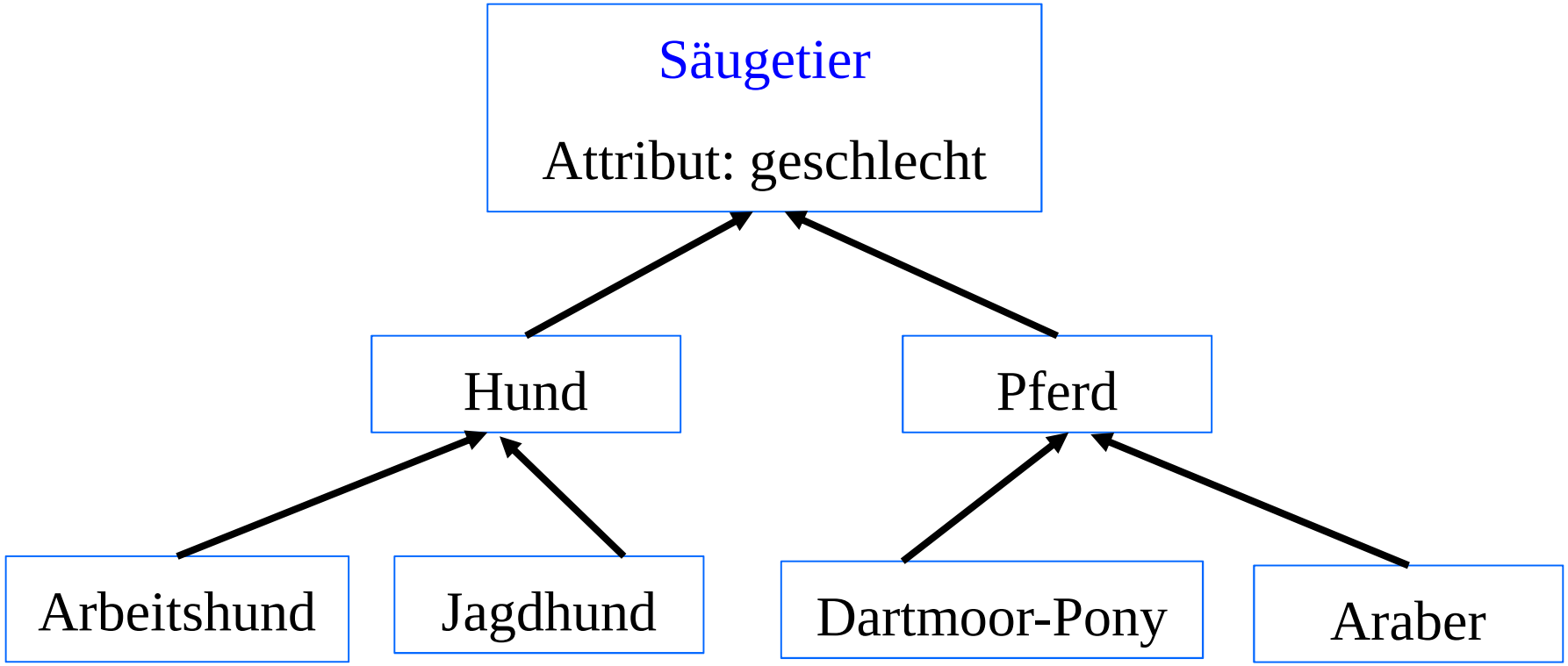
- Eine Klasse sollte nur für eine Sache definiert sein
  - Anfänger bauen manchmal große Klassen
  - Ein Sammelsurium ist schwer zu warten
  - Ein Sammelsurium ist schwer an Kollegen zu vermitteln
  - Beim einem Sammelsurium fragt man sich, **was** macht die Klasse

- Ziel
  - Die Klassen so klein als möglich
  - Die Klassen so groß wie nötig

Objektklassen: Vererbung

- Klassen können von anderen Klassen abgeleitet werden
- **Sie „erben“ alle Attribute**
- **Sie „erben“ alle Methoden**
- Methoden können aber „abgelehnt“ werden
- Man kann „Sub“-Klassen zwingen, Methoden zu implementieren
- Es entsteht keine Code-Redundanz
- Das Schlüsselwort lautet **extends**
- In Java ist nur die Einfach-Vererbung möglich.
  - **Ein** Parent.

Vererbung Beispiel



Objektklassen der Autoverleihfirma

- **Klasse Fahrzeug**
  - Name
  - Anzahl der Räder
- **PKW extends Fahrzeug**
  - PS / KW
  - Farbe
  - Anzahl der Sitze
  - Ausstattung
- **SUV extends PKW**
  - Allrad
- **Cabrio extends PKW**
  - Mit Motorverschluss
- LKW **extends Fahrzeug**
  - Ladefläche
  - LKW-Anhänger
  - zulässiges Anhängergewicht
- Fahrrad **extends Fahrzeug**
  - Mit Motor
  - Höhe

Abstrakte Klassen

- Um Sub-Klassen zu zwingen, Methoden zu implementieren, kann man eine sogenannte abstrakte Klasse definieren
- Eine abstrakte Klasse kann nicht instanziiert werden
- Eine von einer abstrakte Klasse abgeleiteten Klasse **kann** instanziiert werden
- Eine abstrakte Klasse kann auch als Datentyp in einem Array dienen
- Eine abstrakte Klasse kann auch Attribute haben, der Hauptzweck sind aber die Methodendefinitionen

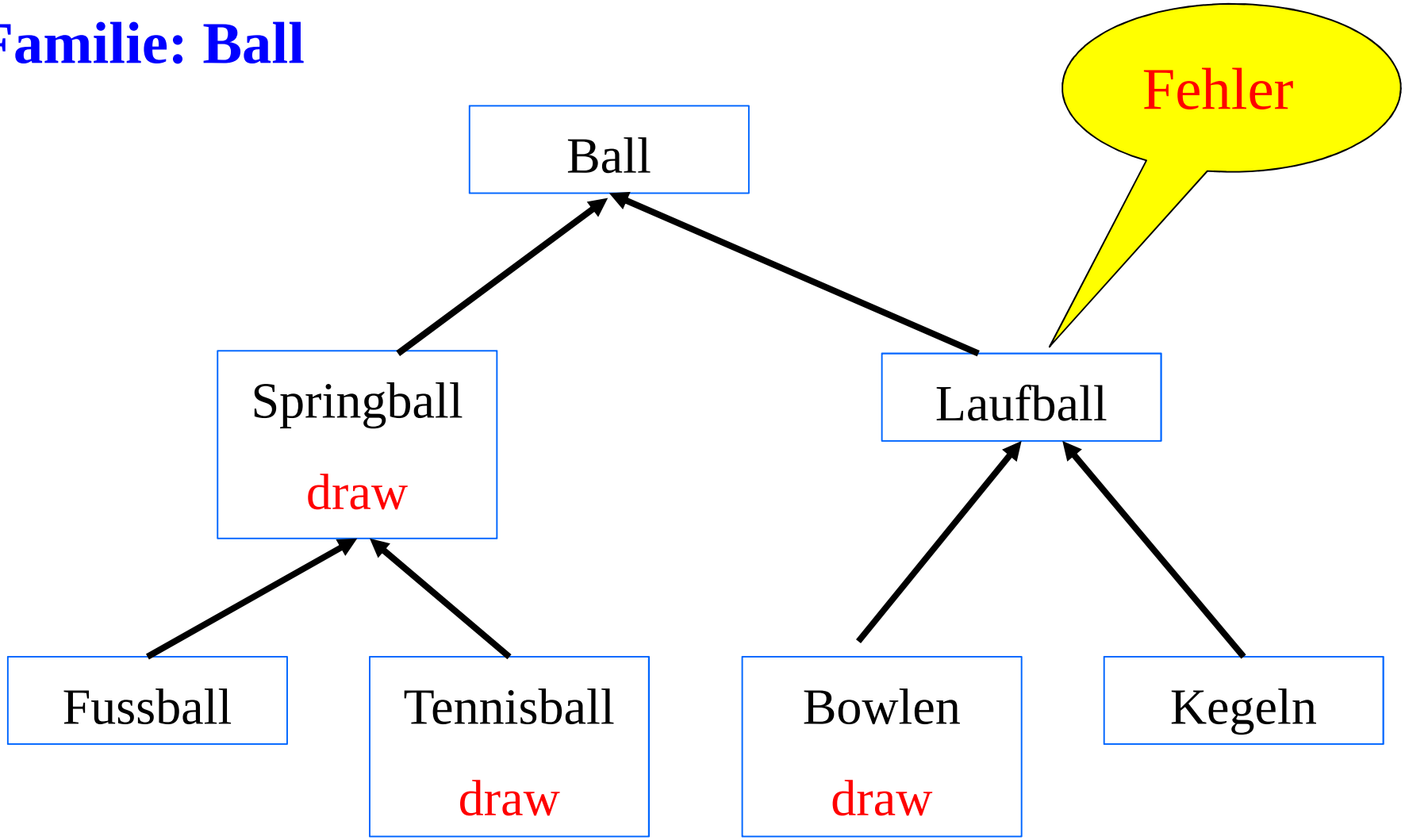
Abstrakte Klassen

Klasse Fahrzeug

- String name
- int anzahlDerRaeder

```
abstract class Fahrzeug {
    private String name;
    private int AnzahlDerRaeder;
    private float preis;

    abstract String getTyp();
    abstract String print();

}
```

Familie: Ball



Polymorphismus, Vielgestaltigkeit

- Abstrakte Methoden müssen immer implementiert werden
- Dynamische Methoden können implementiert werden
- Java sucht im aktuellen Objekt, zum Beispiel „Springball“ nach der gesuchten Methode.
- Wenn Java sie findet, okay
- Falls nicht, sucht Java sie in die nächsthöhere Klasse, bis eine Methode gefunden wurde.

Kann Java auch mal nichts finden ???

Polymorphismus, Vielgestaltigkeit

- Die Polymorphie macht es möglich, dass verschiedene Sub-Klassen dieselbe Methode verstehen, obwohl der technische Aufruf auf diese Anfrage, z. B. draw, völlig unterschiedlich sein kann.
- Auf die Botschaft »draw« können sowohl Objekte vom Typ „Auto“, als auch vom Typ „Fahrrad“ reagieren.
- In jedem Fall wird **eine** Methode aufgerufen.
- Welche entscheidet Java zur Laufzeit.
- Polymorphie wird nun dadurch realisiert, dass ein Objekt eine geerbte Methode, draw, abändern kann, um in der gewünschten Weise zu reagieren. Diesen Vorgang nennt man auch **Überschreiben** einer Methode.
- Überladen ist in **einer** Klasse.
- Überschreiben verwendet mehrere Klassen.

Klassen als Parameter

```
public class Basetest1 {
        public static void message ( String s ) {
          System.out.println( s );
        }
        public static void method (Base b ) {
          message("base" + b.message());
        }
        public static void method (Sub1 s ) {
          message("sub1" + s.message());
        }
        public static void main ( String[] s ) {
          Sub2 s2 = new Sub2 ( ) ;
          Base b = s2 ;
          method ( s2 ) ; // Was wird ausgegeben?, welche Methode wird aufgerufen
          method ( b ) ;
        }
}
```

```
base
sub1
sub2
```

Klassen als Parameter

```
public class Basetest2 {
        public static void main ( String[] s ) {
            Sub s = new Sub();
            s.method (7);              // Was ist die Ausgabe?
            s.method (7.0);            // Was ist die Ausgabe?
            float f = 7.0f;
            s.method (f);              // Und hier ?
       }
}
class Base {
         public void method (int i) {
            System.out.println("Base int");
         }
         public void method (double d) {
           System.out.println( "Base double");
         }
}
```

```
class Sub extends Base {
   public void method (double d) {
       syso("Sub double");
  }
}
```

Interface

Wie kann sichergestellt werden, dass bestimmte Methoden in einer Klasse implementiert sind?

Java bietet den interface-Konstrukt:

- Eine Schnittstelle (engl. Interface) hat einen Namen (Namensgebung wie für Klassen).
- Jedoch enthält ein Interface keine Methodenimplementationen oder Attribute, sondern lediglich die Prototypen der zu implementierenden Methoden.

D.h. ein Interface stellt einen Vertrag über in einer Klasse zu implementierende Methoden dar.

Von einem Interface können **keine Instanzen** erzeugt werden, **aber sehr wohl Referenzen**.

Insbesondere Methodenparameter können Interface-Referenzen sein.

Interface Beispiel

```
class abstract Rechner {
   public int takt;
   public abstract void rechne();
}

interface IRechner {
    public void rechne();
}
```

Interface Beispiel

```
class Notebook extends Rechner {
  public void rechne() {
    // do some stuff
  }
}

class Desktop implements IRechner {
 public void rechne() {
    // do some stuff
  }
}
```

Interface Beispiel

```
class Notebook extends Rechner {
  public void rechne() {
    // do some stuff
  }
}

class Desktop implements IRechner, IDatabase {
 public void rechne() {
    // do some stuff
  }
 public void save2DataBase() {
    // do some stuff
  }
}
```

Interface vs. Vererbung

- Eine Klasse kann nur von einer Oberklasse abgeleitet werden.
- Eine Klasse kann beliebig viele Sub-Klassen haben.
- Eine abstrakte Klasse und ein Interface beschreiben die gewünschte Funktionalität für die Sub-Klassen.
- Eine abstrakte Klasse kann Attribute haben.
- Ein Interface kann Attribute haben, diese sind aber final und static, also Konstanten.
- Die ersten Sub-Klassen müssen die Methoden implementieren.
- Die weiteren Sub-Klassen dürfen die Methoden überschreiben.
- Eine abstrakte Klasse kann nicht erzeugt werden.
- Eine Schnittstelle kann nicht erzeugt werden.
- Eine Klasse kann beliebig viele Interfaces implementieren.

Interface vs. Vererbung

- Jede Schnittstelle definiert eine neue Sicht, eine Art Rolle.
- Implementiert eine Klasse diverse Schnittstellen, können ihre Exemplare in verschiedenen Rollen auftreten.
- Schnittstellen können keine Methoden haben, die Quellcode beinhalten.
- Eine abstrakte Klasse kann eine normale Methode hinzufügen.
- Änderungen in den Sub-Klassen???