



# Programmiermethodik

## Objektorientierte Programmierung

SS 2002

Thomas Kühne

[kuehne@informatik.tu-darmstadt.de](mailto:kuehne@informatik.tu-darmstadt.de)

<http://www.informatik.uni-mannheim.de/informatik/softwaretechnik>

© T. Kühne



## OO-Sprachen

- Urvater: Simula '67
  - » "Algol + Vererbung + Koroutinen"
- Hauptmerkmale
  - » Kapselung (Geheimnisprinzip)
  - » Vererbung (Wiederverwendung)
  - » Polymorphismus (spätes Binden)
- Typische Vertreter
  - » Smalltalk, C++, Eiffel, Java

© T. Kühne



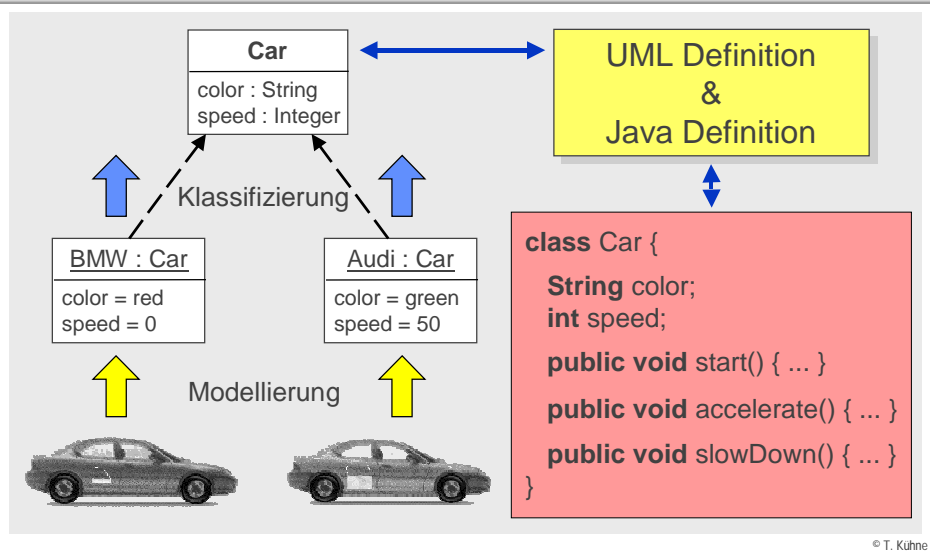
## Java

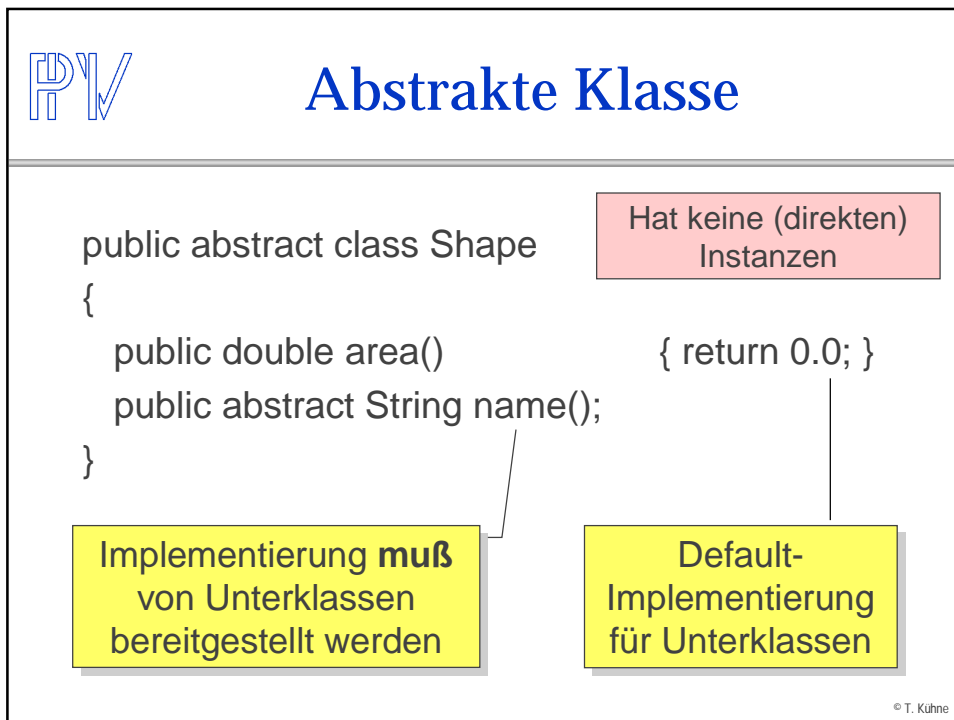
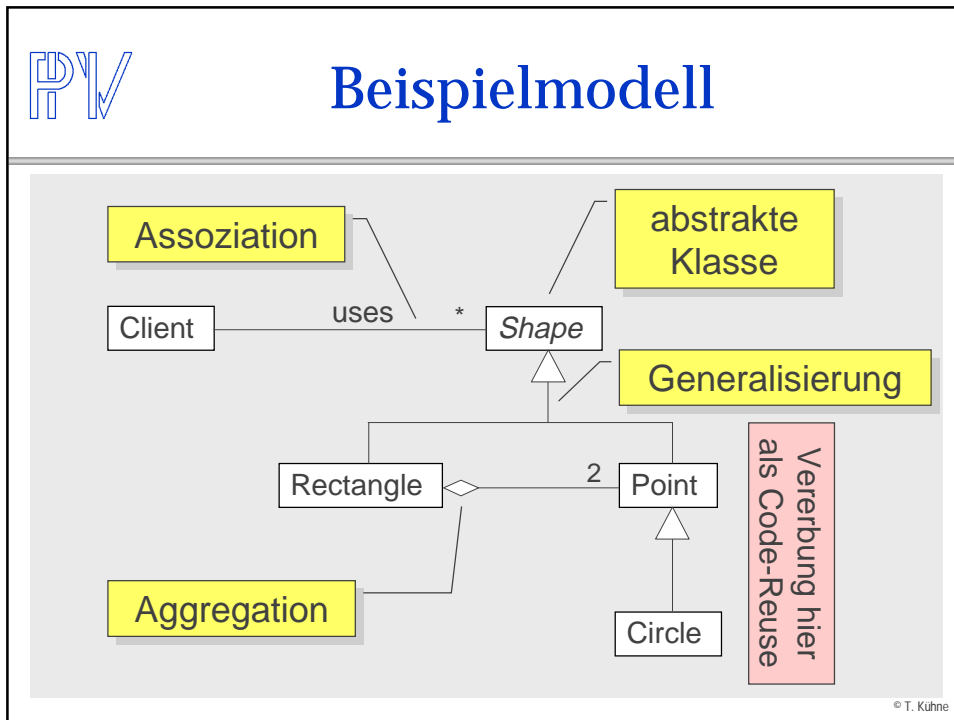
- Einführung durch Sun Microsystems 1996 als WWW Sprache
- Erfolg als "Browser-Sprache" nicht als Software-Engineering Sprache
  - » breite Unterstützung / Sandkastenprinzip!
  - » Schwächen als SE-Sprache
- Bytecode + Virtual Machine
  - » Idee so alt wie Pascal (70'er Jahre) und Smalltalk (dort aber zur Erhöhung der Interaktivität ausgenutzt)

© T. Kühne



## Objekte und Klassen







## Klasse "Point"

```
public class Point extends Shape // inherits from Shape
{
    protected int x, y;           // coordinates of the Point
    public Point( int a, int b )
    {
        x = a;
        y = b;
    }
    public Point() { this( 0, 0 ); }
```

Konstruktor

überladener Konstruktor

© T. Kühne



## Klasse "Point" (fortg.)

```
public int x() { return x; }
public int y() { return y; }
public String name() { return "Point"; }
public String toString()
{
    return "[" + x() + ", " + y() + "];"
}
}
```

nützlich für Konsolen-Ausgabe, aber auch Debugging

© T. Kühne



## Vererbung mit "Circle"

```
public class Circle extends Point // inherits from Point
{
    protected double radius; // additional attribute

    public Circle( double r, int a, int b )
    {
        super( a, b );
        radius = r;
    }

    public Circle() {this ( 0, 0, 0 );}
```

Aufruf des  
Oberklassen-  
Konstruktors

Kreise besitzen alle Merkmale von Punkt  
plus Radius & verändertes Verhalten

© T. Kühne



## Vererbung mit "Circle"

```
public double radius() { return radius; }
public double area() { return Math.PI * radius * radius; }
public String name() { return "Circle"; }

public String toString()
{
    return "Center = " + super.toString() +
        "; Radius = " + radius;
}
}
```

Aufruf der  
Oberklassen-  
Methode

© T. Kühne



## Klasse "Rectangle"

```
public class Rectangle extends Shape
{
    protected Point ul; // upper left corner
    protected Point lr; // lower right corner
    public Rectangle(Point ul, Point lr)
    {
        this.ul=ul;
        this.lr=lr;
    }
}
```

Zugriff auf  
"verdeckte"  
Instanzvariablen

© T. Kühne



## Klasse "Rectangle" (fortg.)

```
public double area()
{
    return Math.abs(lr.x()-ul.x())*Math.abs(lr.y()-ul.y());
}
public String name() { return "Rectangle"; }
public String toString()
{
    return "Rectangle: UL=" + ul + "; RL="+ lr;
}
}
```

automatischer Aufruf von "toString()"

© T. Kühne



## Klasse "Client"

```
import java.awt.Graphics;
import java.applet.Applet;

public class Client extends Applet {
    private Point myPoint;
    private Circle myCircle;
    private Rectangle myRect;
    private Shape arrayOfShapes[];
```

© T. Kühne



## Klasse "Client" (fortg.)

```
public void init()
{
    myPoint= new Point( 7, 11 );
    myCircle= new Circle( 3.5, 22, 8 );
    myRect=new Rectangle(myPoint, new Point(10, 15));
    arrayOfShapes = new Shape[ 3 ];
    arrayOfShapes[ 0 ] = myPoint;
    arrayOfShapes[ 1 ] = myCircle;
    arrayOfShapes[ 2 ] = myRect;
}
```

© T. Kühne



## Klasse "Client" (fortg.)

```
public void paint( Graphics g )
{
    g.drawString( myPoint.name() + ": " +
                  myPoint.toString(), 25, 25 );
    g.drawString( myPoint.name() + ": " +
                  myPoint.toString(), 25, 40 );
    g.drawString( myRect.name() + ": " +
                  myRect.toString(), 25, 55 );
    int yPos = 85;
```

© T. Kühne



## Klasse "Client" (fortg.)

```
for (int i = 0; i < arrayOfShapes.length; i++)
{
    g.drawString( arrayOfShapes[ i ].name() + ": " +
                  arrayOfShapes[ i ].toString(), 25, yPos );
    yPos += 15;
    g.drawString( "Area = " + arrayOfShapes[ i ].area(), 25, yPos );
    yPos += 30;
}
}
```

© T. Kühne





## Klienten-Ausgabe

Point: [7, 11]

Circle: Center = [22, 8]; Radius = 3.5

Rectangle: UL = [7, 11]; LR = [10, 15]

Point: [7,11]

Area = 0

Circle: Center = [22, 8]; Radius = 3.5

Area = 38.4845

Rectangle: UL = [7, 11]; LR = [10, 15]

Area = 12

© T. Kühne

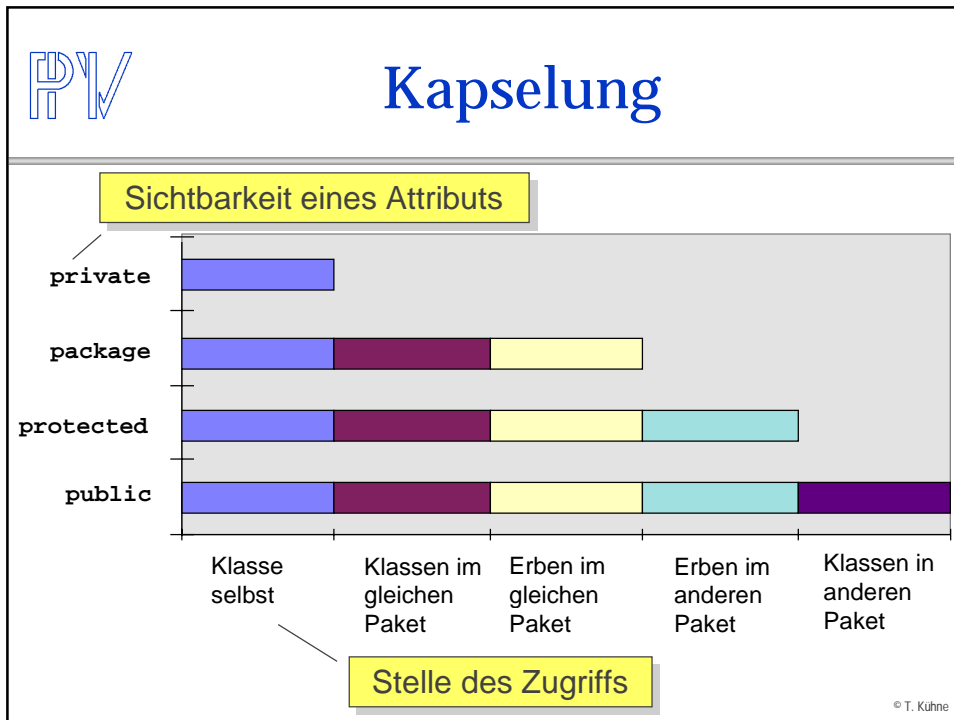


## Kapselung

### Verbergen von Implementierungs- geheimnissen

- Klienten sind vor Veränderungen geschützt
  - » Realisierungsdetails können sich ändern, sollen aber keine Änderungswelle auslösen
- Verwirklichung durch Sichtbarkeitsattribute
  - » public, protected, package, private

© T. Kühne



## Geheimnisprinzip

```

class Auto {
    public Auto( int tMax, String f ) {
        tankgroesse = tMax;
        farbe = f;
        geschwindigkeit = 0;
        . . .
    }
    . . .
    private int tankgroesse,
               geschwindigkeit;
    private String Farbe;
    . . .
}
    
```

} öffentlich

© T. Kühne

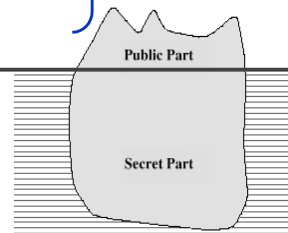


## Geheimnisprinzip

```
class Auto {  
    public Auto( int tMax, String f ) {  
        tankgroesse = tMax;  
        rot = r(f); gruen = g(f); ...  
        geschwindigkeit = 0;  
        . . .  
    }  
    . . .  
}
```

öffentlich

```
private int tankgroesse,  
        geschwindigkeit;  
private int rot, gruen, blau;
```



Änderung der Implementierung nach außen nicht sichtbar.  
Alle Klienten können unverändert weiter verwendet werden



## Vererbung

- Erweitern einer Klasse ohne sie zu ändern
- Attribute und Verhalten werden vererbt
- Neue Attribute und neues Verhalten kann hinzugefügt werden
- geerbtes Verhalten kann überschrieben werden

Face



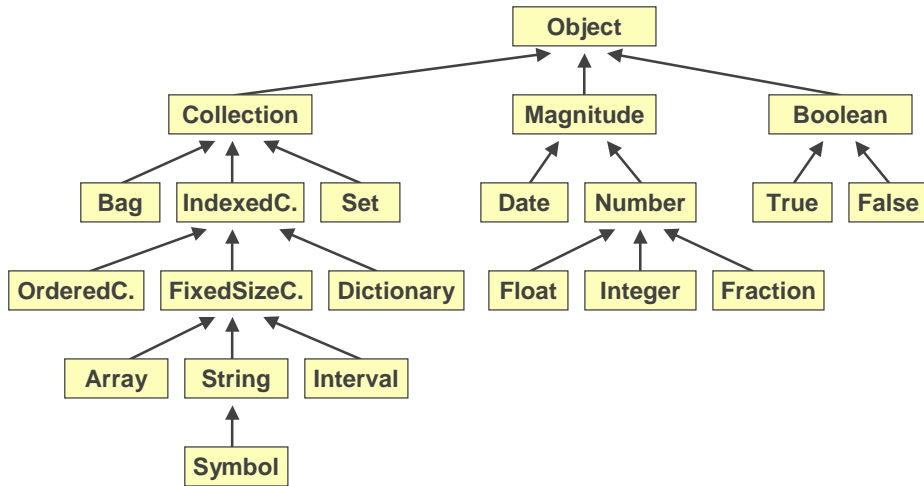
NamedFace



Raging Bull



## Smalltalk Klassenhierarchie



© T. Kühne



## Polymorphismus

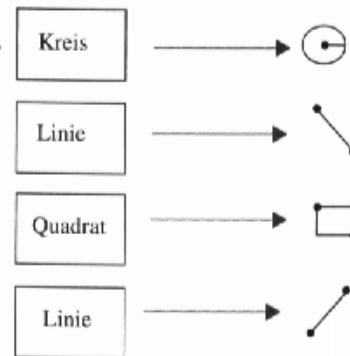
### "Vielgestaltigkeit"

Nachricht von Zeichnung  
an alle Objekte (geom. Figuren)  
**zeichne()**

» Variable eines *statischen*  
Typs kann auf viele  
*dynamische* Objekttypen  
verweisen

- » erlaubt die Formulierung von generischem Code
- » unterstützt Wiederverwendung!

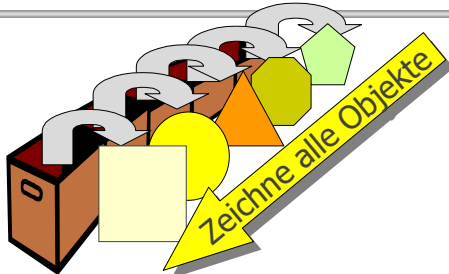
Objekte:



© T. Kühne



## Polymorphismus



- Das übergeordnete Zeichenverfahren (**drawAll()**) kann spezifiziert werden, ohne die aktuellen Typen der Figuren zu kennen.
- Man kann neue Figurtypen später implementieren, Exemplare von ihnen erzeugen und der Liste hinzufügen, ohne das Zeichenverfahren von **drawAll()** anpassen zu müssen.
- Der Dienstabnehmer (**Client**) ist unabhängig vom Dienstanbieter (die konkreten **Shape** Unterklassen).

© T. Kühne



## Java Schnittstellen

### Was tun wenn ein Objekt in mehreren Kontexten einsetzbar ist?

- Eine Klasse kann zwar nur eine direkte Oberklasse besitzen (**Einfachvererbung**)
- Aber: Eine Klasse kann mehrere **Schnittstellen** implementieren
- Klassen, die Schnittstellen implementieren, können eigene, zusätzliche Operationen definieren

Beispiel:  
Amphibienfahrzeug  
kann bei Land  
(`run()`) und  
Wasserrennen  
(`swim()`) mitmachen.

© T. Kühne



## Java Schnittstellen

```
interface DisplayShape {  
    public void draw();  
}
```

statt "extends"!

```
class Circle implements DisplayShape {  
    public void draw() {...};  
    public void area() {...};  
}
```

© T. Kühne



## Java Schnittstellen

```
interface DisplayShape {  
    public void draw();  
}
```

```
interface GeometricShape {  
    public double area();  
}
```

mehrere  
Schnittstellen

```
class Circle implements DisplayShape,  
                        GeometricShape {  
    public void draw() {...};  
    public void area() {...};  
}
```

© T. Kühne



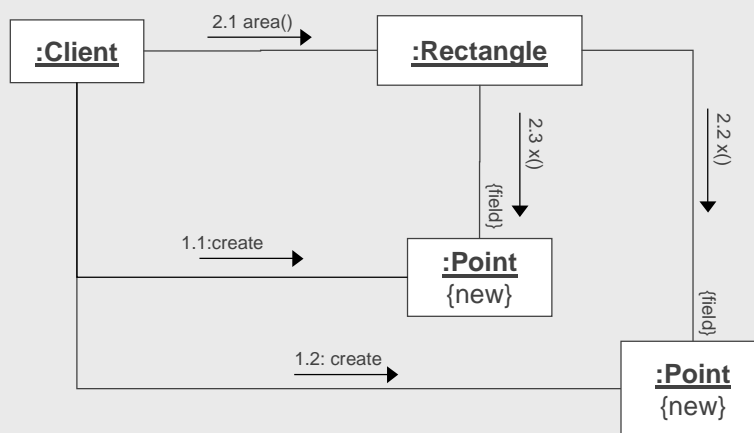
## Java Schnittstellen

```
class Main {  
    public static void main(String[] args)  
    {  
        GeometricShape circle = new Circle(...);  
        GeometricShape rect = new Rectangle(...);  
        GeometricShape [] gsList =  
            new GeometricShape [2];  
        gsList[0] = circle;  
        gsList[1] = rect;  
        for (int i=0; i<gsList.length; i++)  
            System.out.println(g[i].area());  
    }  
}
```

© T. Kühne



## Nachrichten



© T. Kühne



## Methodenausführung

```
figs.add(face); figs.add(bull); figs.add(hexagon);
figs.drawAll();
```



Raging Bull



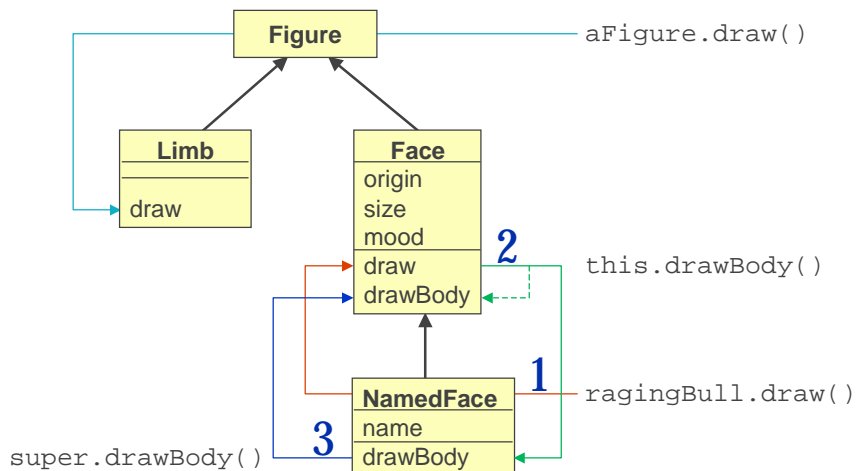
### Ausführung von

- einer von mehreren Unterklassenmethoden
- einer Eltern Methode
- Eltern-Methode aus Kind-Methode heraus
- Kind-Methode statt Eltern-Methode

© T. Kühne



## Methodenausführung



© T. Kühne





## Nachrichten-"Kanäle"

- ① **Wert eines Attributs**  
Instanzvariable (Attribute) enthält die Referenz auf das Empfängerobjekt
- ② **Aktueller Parameterwert einer Methode**  
Objektreferenz wird als Argument an eine Methode übergeben
- ③ **Lokal erzeugtes Objekt**  
Objekt wird in einer Methode erzeugt (temporäre Variablennutzung optional) und dann als Empfänger genutzt

© T. Kühne



## Nachrichten-"Kanäle"

- ④ **Wert eines Operationsaufrufs**  
Der Rückgabewert einer Methode ist eine Referenz auf das Empfängerobjekt
- ⑤ **Wert eines Klassenattributs**  
Eine (static) Klassenvariable hält die Empfängerobjektreferenz

© T. Kühne

PV Nachrichten-"Kanäle"

```
class Client {  
    ServerClass1 server1;  
  
    public Client() { server1 =  
        new ServerClass1(); }  
  
    public void op(ServerClass2 serv2) {  
  
        server1.do_smth(...)  
        serv2.do_smth_else(...)  
        ServerClass4 serv4 =  
            new ServerClass4();  
  
        (server1.result()).message();  
    }  
}
```

Attribut

Argument

Lokales Objekt

Rückgabewert

© T. Kühne

PV Java – Der letzte Schrei?

...all the speed of an interpreter combined with the "convenience" of an extra compilation step

primitive types make it a hybrid language

(still) no generic types

*Java is a joke, only it's not funny*

Alan Lovejoy

© T. Kühne