Objekte und Objekttypen in der Programmierung oder OOP in 10 Konzepten

Jürgen F H Winkler
Friedrich-Schiller-Universität Jena
Institut für Informatik
psc.informatik.uni-jena.de
Jena, 3. März 2006

© J F H Winkler, 2006

Objekte und Objekttypen

- Objekt? zusammengesetzte Grösse im Bereich der Programmierung (spez. Bedeutung von „Objekt“)

- Objekttyp? entsprechender Typ i.S. von „Typ“ in Programmen

n Personen verwenden in der Regel n+1 verschiedene OOP Nomenklaturen

„Mit **Objekt** werden Gaststätten und Ferien bzw. Erholungseinrichtungen bezeichnet. Ein Objektleiter ist zB der Leiter einer **...**“

Sabina Schroeter, 1994

© J F H Winkler, 2006

Objekte und Objekttypen

n Personen verwenden in der Regel n+1 verschiedene

OOP Nomenklaturen

„OOP …

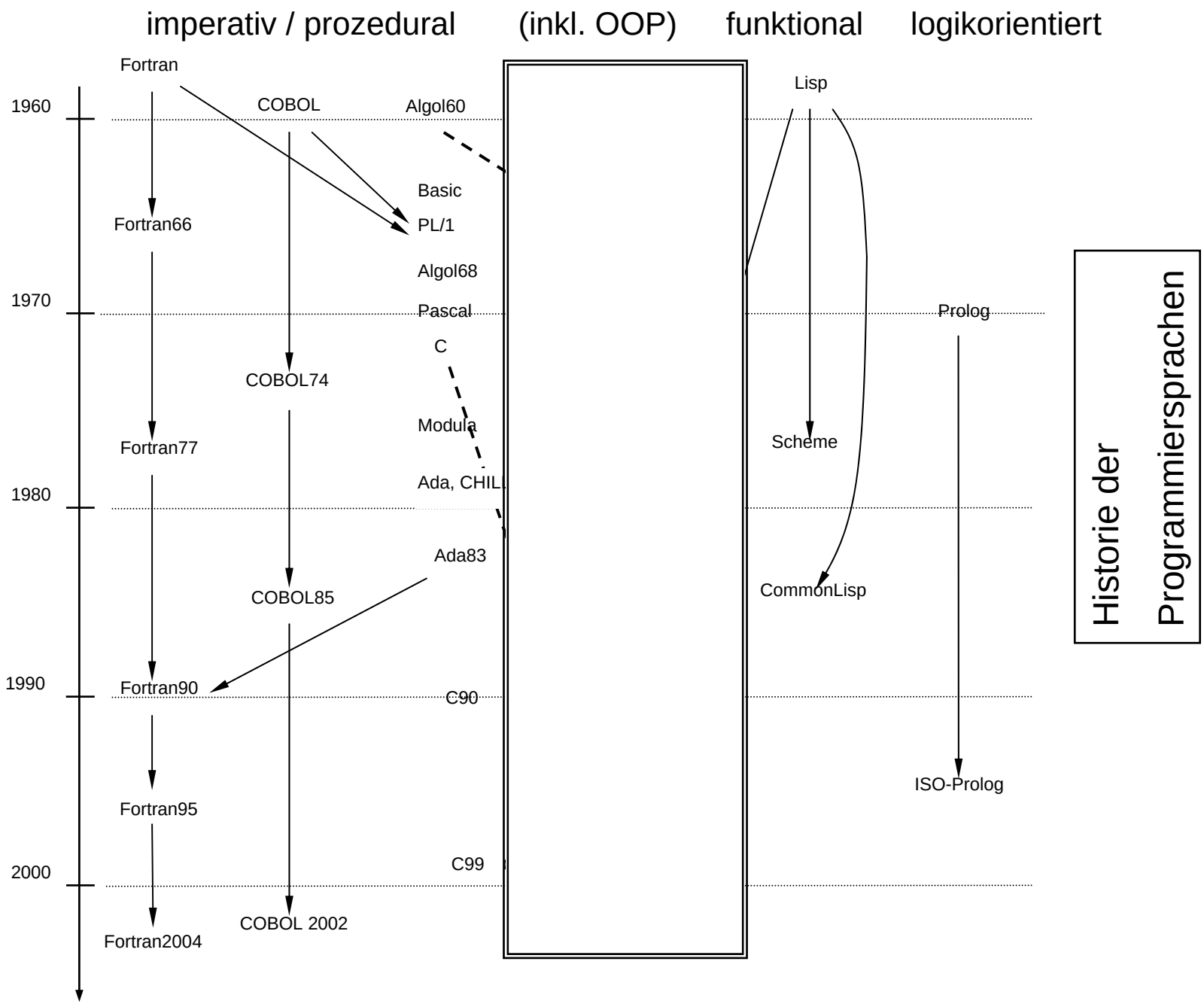
Moderne prozedurale Sprachen verfügen dabei über Sprachelemente, die über Variablen von einem Prozedurtyp die Zuweisung von Verarbeitungsalgorithmen in der Laufzeit erlauben (dynamische Bindung)“

Engelmann, Lutz (Hrsg.): Informatik bis zum Abitur. Paetec 2002

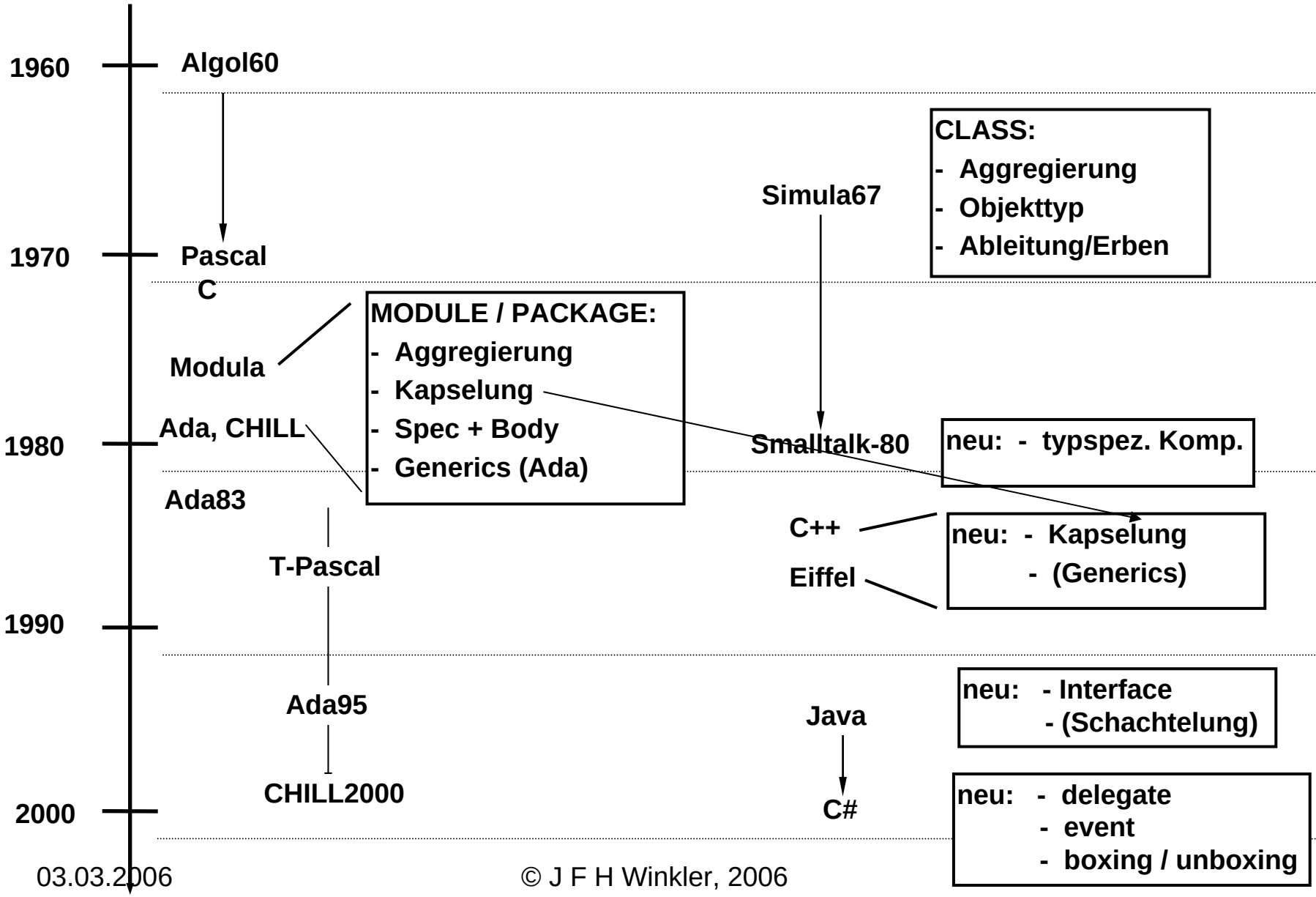
© J F H Winkler, 2006

- Historie der Programmiersprachen und der OOP
- Einzeltyp und Einzelobjekt
- Abgeleiteter Typ
- Zusammenfassung

© J F H Winkler, 2006


imperativ / prozedural
(inkl. OOP)
funktional
logikorientiert
Fortran
COBOL
Algol60
Lisp
1960
Basic
PL/1
Fortran66
Algol68
1970
Pascal
Prolog
C
COBOL74
Modula
Fortran77
Scheme
Ada, CHIL
1980
Ada83
COBOL85
CommonLisp
1990
Fortran90
C90
ISO-Prolog
Fortran95
2000
C99
COBOL 2002
Fortran2004
Historie der Programmiersprachen

Historie der Konzepte



© J F H Winkler, 2006

Einzeltyp und Einzelobjekt

© J F H Winkler, 2006

```
                                                -- Typdeklaration (Ada)
TYPE  PunktTy is RECORD
                        xPos            : xBereichsTy;
                        yPos            : yBereichsTy;
                        Farbe           : FarbTy;
                        Linienstaerke   : LinienTy;
                    END;
```

– Repräsentation von Zeichenpunkten in einem Programm

```
    -- Deklaration von Variablen / Instanzen des Typs „PunktTy“

VAR  P1, P2 : PunktTy :=  ( xPos          => 0.0,
                            yPos          => 0.0,
                            Farbe         => gelb,
                            Linienstaerke => 1.0 );

-- Struktur der Instanzen durch Typdeklaration festgelegt
```


© J F H Winkler, 2006

```
                                    --  Deklaration von Verschiebeoperation

PROCedure MoveTo(P: in out PunktTy;
                 x: in xBereichsTy;
                 y: in yBereichsTy);
BEGIN
    P.xPos := x;
    P.yPos := y;
END MoveTo;



                                                    -- Manipulation

 MoveTo(Punkt => P1, x => 2.0, y=> -3.5);
```


© J F H Winkler, 2006

Programmstruktur

```
TYPE  PunktTy  is …

VAR  P1, P2 : PunktTy := …

PROCedure MoveTo( … );
```

noch recht übersichtlich

© J F H Winkler, 2006

nun zusätzlich RechteckTy

```
TYPE  PunktTy is …
VAR  P1, P2 : PunktTy …
PROCedure MoveTo( … )
TYPE RechteckTy is …
VAR R1, R2: RechteckTy …
PROCedure MoveTo( … )
```

noch recht übersichtlich:

Zusammengehöriges steht noch zusammen

© J F H Winkler, 2006

nun aber in (Standard) Pascal:

Sortenreihenfolge: LABEL, CONST, TYPE, VAR, PROC

```
TYPE  PunktTy
TYPE RechteckTy
VAR P1, P2 : PunktTy
VAR R1, R2: RechteckTy
PROCedure MoveTo
PROCedure MoveTo
```

unübersichtlich

Zusammengehöriges nun getrennt

© J F H Winkler, 2006

Wie würde das erst bei 6 Typen mit je 5 Operationen aussehen?

(in z.B. Borland-Pascal könnte man pro Typ ein Unit bilden)

Lösung in OOP noch besser:

auch Operationen innerhalb der Typdeklaration deklarieren

© J F H Winkler, 2006

```
                                                    --  Objekttypdeklaration
TYPE  PunktTy = OBJECT                              --  freie Syntax
      xPos            : xBereichsTy := 0.0;
      yPos            : yBereichsTy := 0.0;
      Farbe           : FarbTy := gelb;
      Linienstaerke   : LinienTy := 1.0;
      PROC MoveTo(x: in xBereichsTy;   y: in yBereichsTy)
            xPos := x;
            yPos := y;
      END MoveTo;
END  PunktTy;
```

Repräsentation von Zeichenpunkten in einem Programm

```
    -- Deklaration von Variablen / Instanzen des Typs „PunktTy“

VAR  P1, P2 : PunktTy;
          -- Struktur der Instanzen durch Typdeklaration festgelegt
P1.MoveTo(x => 2.0, y => -3.5);                  -- Manipulation
```


© J F H Winkler, 2006

```
TYPE  PunktTy = OBJECT
      xPos            : xBereichsTy := 0.0;
      yPos            : yBereichsTy := 0.0;
      Farbe           : FarbTy := gelb;
      Linienstaerke   : LinienTy := 1.0;
      PROC MoveTo(x: in xBereichsTy;   y: in yBereichsTy)
            xPos := x;
            yPos := y;
      END MoveTo;
END  PunktTy;
```

PunktTy	: ein Objekt-Typ (auch „Klasse“ genannt)
xPos, yPos, …	: Komponenten (auch „members“ genannt)
xPos	: Variable (auch „field“ genannt)
MoveTo	: Prozedur (auch „method“ genannt)

© J F H Winkler, 2006

```
-- Deklaration von Variablen / Instanzen des Typs „PunktTy“
VAR  P1, P2 : PunktTy;
        -- Struktur der Instanzen durch Typdeklaration festgelegt
P1.MoveTo(x => 2.0, y => -3.5);                -- Manipulation
```

P1 : Variable / Objekt / Instanz vom Typ „PunktTy“

P1.MoveTo(x => 2.0, y => -3.5);
: Aufruf von MoveTo an der Instanz P1

P2.MoveTo(x => 2.0, y => -3.5);
: Aufruf von MoveTo an der Instanz P2

© J F H Winkler, 2006

-- Objekttypdeklaration

```
class  PunktTy {
        float  xPos             = 0.0;
        float  yPos             = 0.0;
        FarbTy  Farbe           = gelb;
        float  Linienstaerke    = 1.0;
        void MoveTo(float x, float y) {
                xPos = x;
                yPos = y;
        } // MoveTo
}  // PunktTy
```

C#

```
                -- Deklaration von Variablen / Instanzen des Typs „PunktTy“
PunktTy  P1 = new PunktTy( ), P2;
                        -- Struktur der Instanzen durch Typdeklaration festgelegt
P1.MoveTo(2.0, -3.5);                                   -- Manipulation
```


© J F H Winkler, 2006

1. Grundkonzept der OOP

Aggregierung

Zusammenfassung unterschiedlicher Komponenten,
insbesondere auch Unterprogramme,
zu einer manipulierbaren Grösse

Typisch:
Zusammenfassung inhaltlich zusammengehöriger Komponenten

© J F H Winkler, 2006

2. Grundkonzept der OOP

Enge Kopplung

Die Komponenten sind eng gekoppelt,
da sie aufeinander Bezug nehmen können
=> Zugriff auf xPos in MoveTo

Solch ein Zugriff ist z.B. bei einem
Modula-Record mit Prozedurkomponenten
nicht so möglich

© J F H Winkler, 2006

```
class  PunktTy {
        float  xPos             = 0.0;
        float  yPos             = 0.0;
        FarbTy  Farbe           = gelb;
        float  Linienstaerke  = 1.0;            //  Bereich 0.5 .. 5.0
        void MoveTo(float x, float y) {
                xPos = x;
                yPos = y;
        } // MoveTo
}  // PunktTy
```

```
PunktTy  P1 = new PunktTy( );

P1.Linienstaerke = -3.0;          //   ???  unzulaessiger Wert
```


© J F H Winkler, 2006

3. Grundkonzept der OOP

Kapselung

Komponenten können nach aussen verborgen werden,
um zu vermeiden, dass eine Instanz in einen unzulässigen Zustand versetzt wird, d.h. einen unzulässigen Wert erhält.
=> mehrere Exportstati

PunktTy:

Mögliche Werte: MW = float × float × FarbTy × float

Zulässige Werte: ZW = float × float × FarbTy × [0.5, 5.0]

$ZW \subset MW$ kommt häufig vor

© J F H Winkler, 2006

```
class  PunktTy {
    protected  float  xPos           = 0.0;     //  nichtöffentlich
    protected  float  yPos           = 0.0;
    protected  FarbTy  Farbe         = gelb;
    protected  float  Linienstaerke = 1.0;
    public void MoveTo(float x, float y) {      //  öffentlich
            xPos = x;
            yPos = y;
    } // MoveTo
}  // PunktTy
```

```
PunktTy  P1 = new PunktTy( );
P1.Linienstaerke = -3.0;          //  illegaler Zugriff
                                  //  Fehler zur Übersetzungszeit
P1.MoveTo(2.0, 4.0);              //  OK  MoveTo ist öffentlich
```

aber: intern sind immer noch Fehlzuweisungen möglich

© J F H Winkler, 2006

```
class  PunktTy {
    protected  float  xPos           = 0.0;     //  nichtöffentlich
    protected  float  yPos           = 0.0;
    protected  FarbTy  Farbe         = gelb;
    protected  float  Linienstaerke = 1.0;
    public void MoveTo(float x, float y) {      //  öffentlich
            xPos = x;
            yPos = y;
    } // MoveTo
}  // PunktTy
```

public:	öffentliche Komponente
	ausserhalb von PunktTy mit “Instanz . Komp-Bez“ zugreifbar
	P1.MoveTo(2.0, 4.0);
protected:	nichtöffentliche Komponente
	ausserhalb von PunktTy nicht zugreifbar

© J F H Winkler, 2006

4. Grundkonzept der OOP

Manipulierbarkeit

Instanzen können wie andere Werte (z.B. Arrays) manipuliert werden:

Zuweisung:	P1 := P2;
Parameter:	PROC Q(Punkt: in PunktTy; …);
Vergleich:	IF P1 = P2 THEN …
Ziel von Verweisen:	TYPE VerwPunktTy is ACCESS PunktTy;

Ein Modula-Modul kann z.B. nicht als Parameter verwendet werden.

© J F H Winkler, 2006

Objekte und Objekttypen

```
class  PunktTy {
    protected  float  xPos            = 0.0;
    protected  float  yPos            = 0.0;
    protected  FarbTy  Farbe          = gelb;
    protected  float  Linienstaerke = 1.0;
    public void MoveTo(float x, float y) {
            xPos = x;     yPos = y;
    } // MoveTo
}  // PunktTy
```

Problem: wie eine PunktTy-Instanz in den Zustand
(xPos=>2.0, yPos=>14.5, Farbe=>blau, Linienstaerke=>2.3)
bringen?

```
PunktTy  P1 = new PunktTy( );
P1.MoveTo(2.0, 14.5);
P1.SetFarbe(blau);
P1.SetLinienStaerke(2.3);
```

Wirkt umständlich und mühsam

© J F H Winkler, 2006

5. Grundkonzept der OOP

Konstruktor

Prozedur welche in der Deklaration einer Variablen aufgerufen werden kann.

Hauptzweck: Initialisierung der Variablen / Instanz

© J F H Winkler, 2006

```
class  PunktTy {
    protected  float  xPos            = 0.0;  . . .
    public void MoveTo(float x, float y) {  . . . }
    public  PunktTy(float x, float y, FarbTy f, float ls) {
            xPos = x;
            xPos = y;                  //  Konstruktor, heisst wie der Typ
            Farbe = f;
            Linienstaerke = ls;  }
}  // PunktTy
```

```
PunktTy  P1 = new  PunktTy(2, 14.5, blau, 2.3 );
```

Wesentlich einfacher

© J F H Winkler, 2006

Was ist ein Objekt?

hängt von der Art des Objekttyps ab

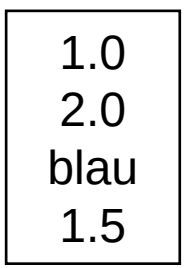
zwei Arten: Wert-Typ (value type) : C++

Referenz-Typ (reference type) : C#, Java, Smalltalk

C++

```
PunktTy  P1(1.0, 2.0, blau, 1.5);
```

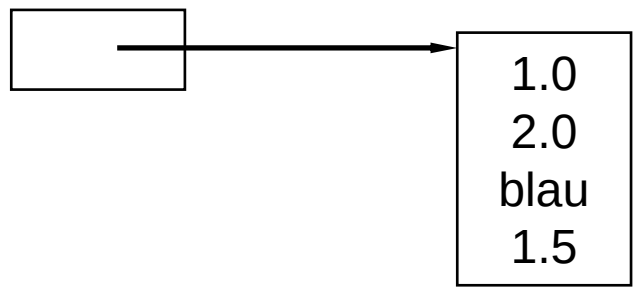
P1



C#

```
PunktTy  P1 =
        new PunktTy(1.0, 2.0, blau, 1.5);
```

P1



© J F H Winkler, 2006

Begriffe

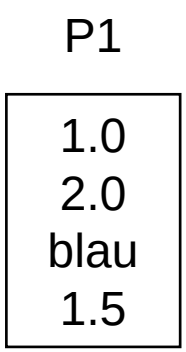
C++

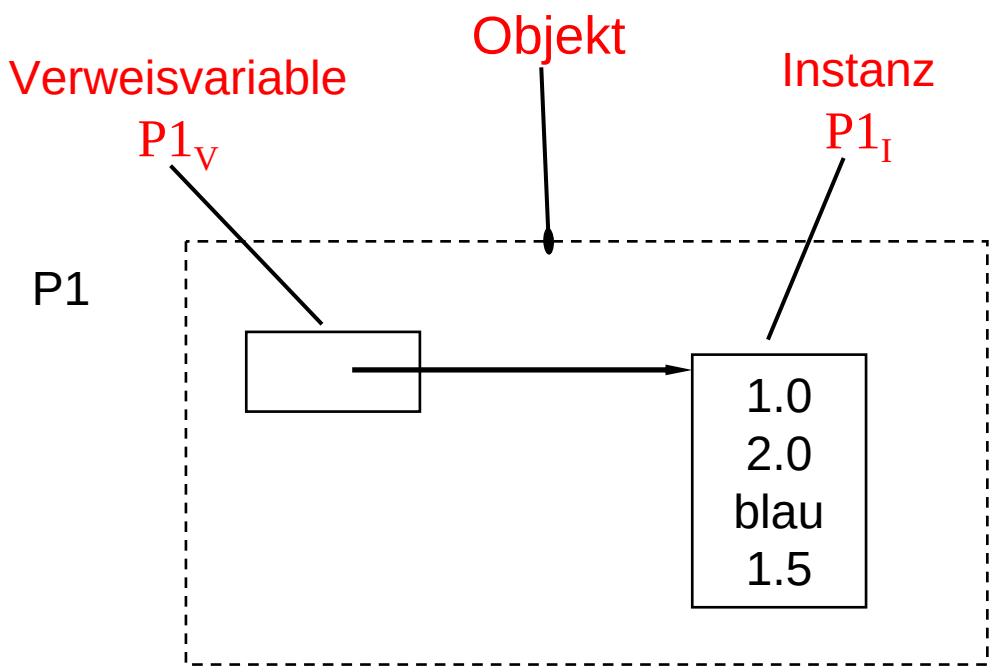
```
PunktTy  P1(1.0, 2.0, blau, 1.5);
```

C#

```
PunktTy  P1 =
        new PunktTy(1.0, 2.0, blau, 1.5);
```

Objekt = Instanz





© J F H Winkler, 2006

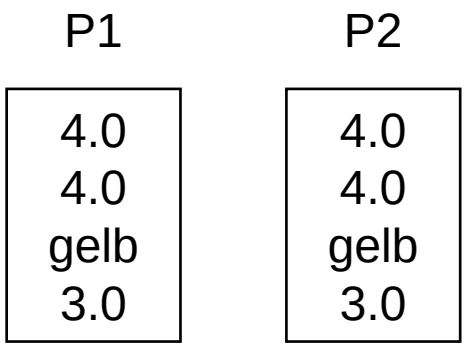
Gravierende Unterschiede bei Zuweisung und Vergleich

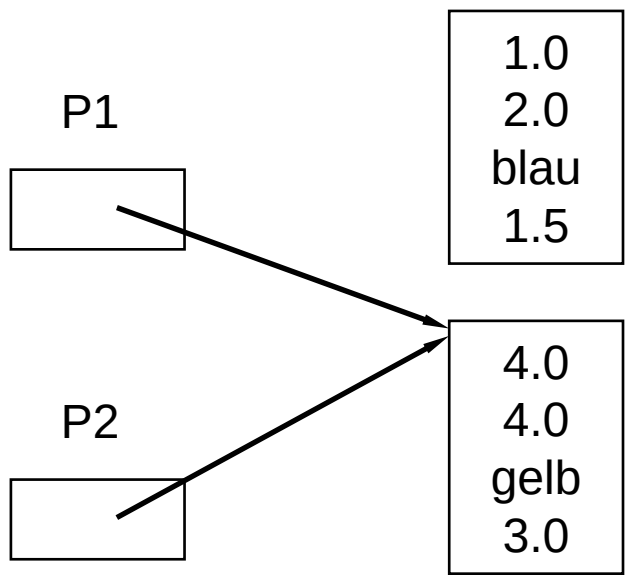
```
PunktTy  P1(1.0, 2.0, blau, 1.5);
PunktTy P2(4.0, 4.0, gelb, 3.0);

P1 = P2;        // Zuweisung
```

```
PunktTy  P1 =
        new PunktTy(1.0, 2.0, blau, 1.5);
PunktTy  P2 = new PunktTy(4.0, …);

P1 = P2;
```





P2.move(5.0, 5.0)

6. Grundkonzept der OOP

Objekt

Wert-Typ : Objekt = Instanz

Referenz-Typ : Objekt = Verweis + Instanz

Objekt-Typ ist impliziter Verweistyp

© J F H Winkler, 2006

Instanzspezifische vs typspezifische Komponenten

instanzspezifisch : werden in jeder Instanz repliziert

typspezifisch : sind nur einmal „im Typ“ vorhanden

Bisherige Komponenten waren alle instanzspezifisch

Neue typspezifische Komponente: NullPunkt (0.0, 0.0, schwarz, 1.0)

© J F H Winkler, 2006

```
class  PunktTy {
    protected  float  xPos  = 0.0;  . . .

    public static readonly PunktTy  NullPunkt =
                                new PunktTy(0.0, 0.0, schwarz, 1.0);

    public void MoveTo(float x, float y) {  . . . }
    public PunktTy(float x, float y, FarbTy f, float ls) {
            xPos = x;
            . . .  }
}  // PunktTy
```

```
PunktTy  P1  =  PunktTy . NullPunkt;      //  Zugriff über Typ
```


© J F H Winkler, 2006

7. Grundkonzept der OOP

instanzspezifische
und
typspezifische Komponeten

Variable, Unterprogramme, Objekttypen

Zugriffsregel: im typspezifischen Teil der Objekttypdeklaration darf nur auf typspezifische Komponenten zugegriffen werden

© J F H Winkler, 2006

Instanzspezifische vs typspezifische Komponenten

=> Janusgesicht des Objekt-Typs

Bezüglich seiner instanzspezifischen Komponenten verhält sich ein Objekttyp wie eine Schablone für die Instanzen

Bezüglich seiner typspezifischen Komponenten verhält sich ein Objekttyp wie ein Modul / Package / Unit.

© J F H Winkler, 2006

Typableitung

© J F H Winkler, 2006

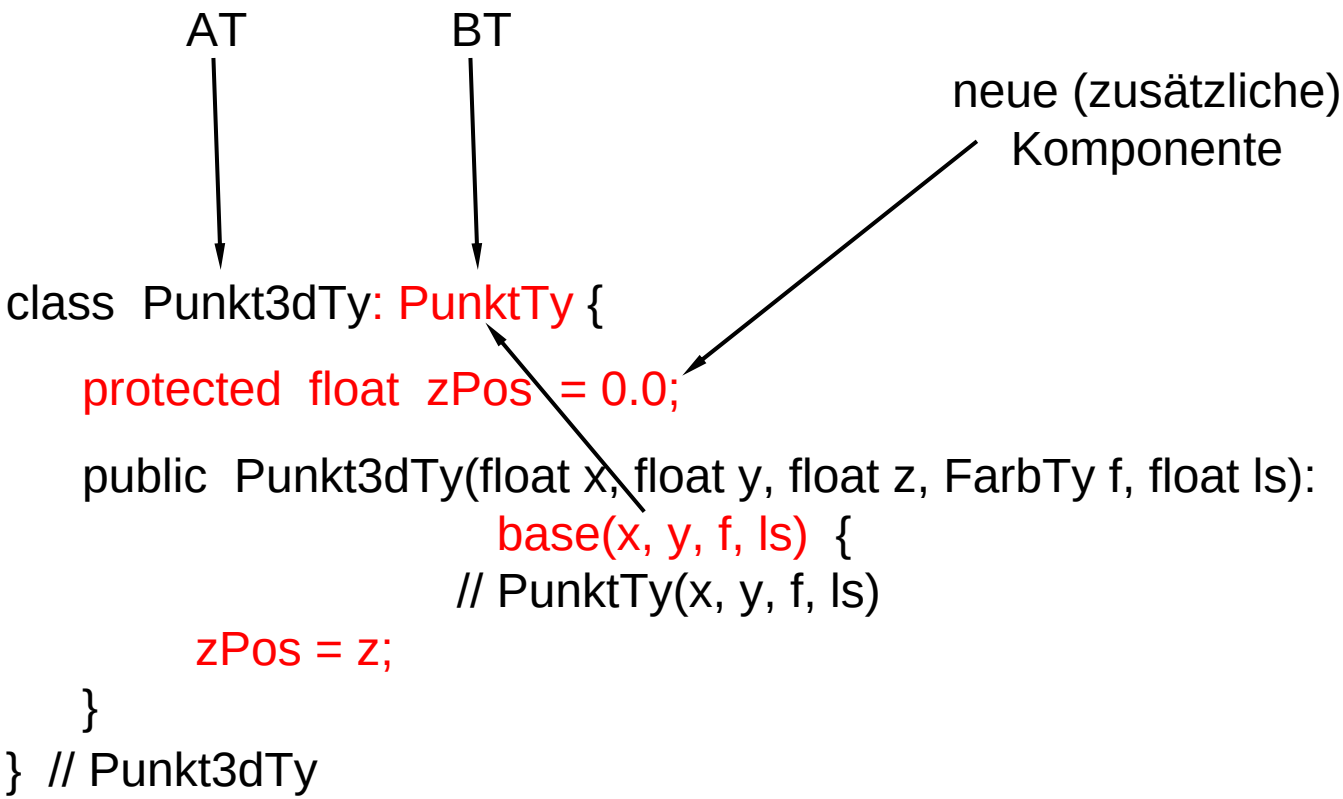
Typableitung: auf der Basis eines vorhandenen Objekttyps BT einen neuen Objekttyp AT deklarieren, so dass AT tendenziell eine Erweiterung von BT ist.

Beispiel: Punkt im 3-dim Raum: zusätzliche Komp. zPos

Grundregel: im abgeleiteten Typ müssen lediglich die neuen Komponenten deklariert werden.

Ausserdem erbt AT die meisten Komponenten von BT.

© J F H Winkler, 2006



© J F H Winkler, 2006

Interne Wirkung der Ableitungsklausel:

es wird so getan,

als seien die erbbaren Komponenten von BT

auch in AT deklariert

© J F H Winkler, 2006

```
class  PunktTy {
      protected  float  xPos  = 0.0;  . . .
      public void MoveTo(float x, float y) {  . . . }
      public void PunktTy(float x, float y, FarbTy f, float ls) {
            xPos = x; . . .  }
} // PunktTy

class  Punkt3dTy: PunktTy {
     //  protected  float  xPos  = 0.0;  . . .
     protected  float  zPos  = 0.0;
     //  public void MoveTo(float x, float y) {  . . . }
     public void Punkt3dTy(float x, float y, float z, FarbTy f, float ls):
                                  base(x, y, f, ls)  {
          zPos = z;  }
} // Punkt3dTy
```

Konstruktoren sind häufig nicht erbbar

© J F H Winkler, 2006

Vorteil der Ableitungsklausel:

kein Eingriff in bestehende SW

AT bezieht sich zwar auf BT

BT bleibt ansonsten aber unberührt

© J F H Winkler, 2006

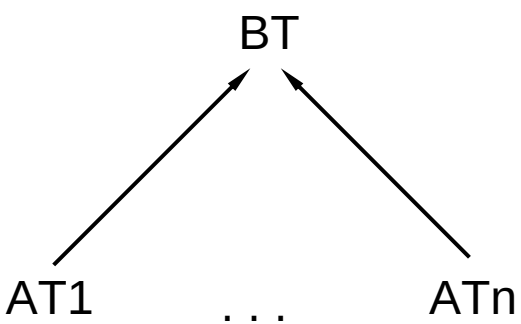
Externe Wirkung der Ableitungsklausel:

Deklaration von verwandten Typen

ähnlich wie bei Records mit Varianten

aber weniger eng gekoppelt

=> grössere Flexibilität



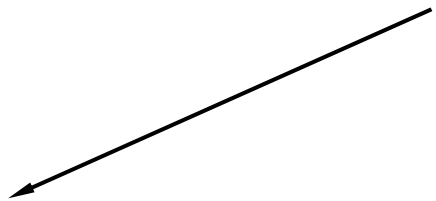

```
TYPE  BAT = RECORD
   BT-Komponenten
   CASE Steuerkomp: Txy  OF
      at1:  AT1-Komponenten
      . . .
      atn:  ATn-Komponenten
   END;
```

8. Grundkonzept der OOP

Typableitung

und

Erben von Komponenten



erben : inherit

vererben : bequeath

© J F H Winkler, 2006

9. Grundkonzept der OOP

Reimplementierung von Unterprogrammen

für ein geerbtes Unterprogramm kann
in AT ein neuer Rumpf angegeben werden
d.h. keine neue (zusätzliche) Komponente
sondern Modifikation einer geerbten Komponente

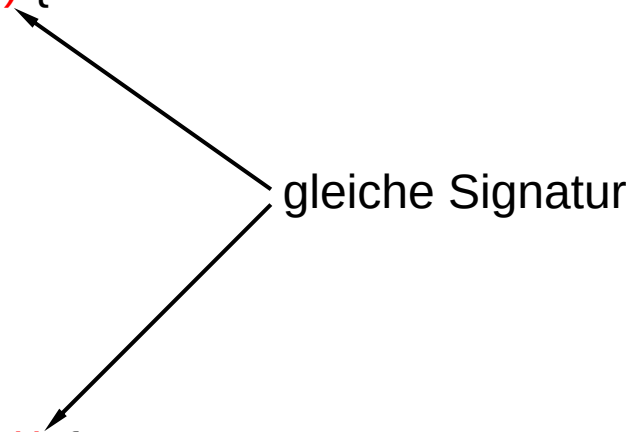
engl: override = deutsch ???

Beispiel: Operation „Spiegeln“ für Punkte

© J F H Winkler, 2006

```
class  PunktTy {
     public virtual void Spiegeln( ) {
          xPos = -xPos;
          yPos = -yPos;  }
     }  // PunktTy


class  Punkt3dTy: PunktTy {
     public override void Spiegeln( ) {
          base.Spiegeln( );
          z.Pos = -zPos; }
     }  // Punkt3dTy
```



10. Grundkonzept der OOP

Polymorphismus von Verweisvariablen

Eine Verweisvariable mit statischem Typ BT darf
auf Instanzen von BT und
auf Instanzen von aus BT abgeleiteten Typen
verweisen.

Gilt bei: impliziten Verweistypen
expliziten Verweistypen gleichermaßen

© J F H Winkler, 2006

Polymorphismus und Reimplementierung

```
class  PunktTy {
     public virtual void Spiegeln( ) {
          xPos = -xPos;
          yPos = -yPos;  }
     }  // PunktTy
```



```
class  Punkt3dTy: PunktTy {
     public override void Spiegeln( ) {
          base.Spiegeln( );
          z.Pos = -zPos; }
     }  // Punkt3dTy
```

```
PunktTy  P1 =
        new Punkt3dTy( …);
P1.Spiegeln( );
        //  Punkt3dTy:Spiegeln( )
P1 = new PunktTy( … );
P1.Spiegeln( );
        //  PunktTy:Spiegeln( )
```

=> es wird der zur aktuellen Instanz gehörende Rumpf ausgeführt, d.h. der jeweils passende Rumpf

© J F H Winkler, 2006

Polymorphismus und Reimplementierung

```
PunktTy  P1;   //  P1 = null
if  (Bedingung)
    P1 = new PunktTy( … );
else
    P1 = new Punkt3dTy( … );

P1.Spiegeln( );
```

=> es wird der zur aktuellen Instanz gehörende Rumpf ausgeführt, d.h. der jeweils passende Rumpf

© J F H Winkler, 2006

Zusammenfassung

© J F H Winkler, 2006

Einzeltyp / -objekt

- Aggregierung
- Enge Kopplung
- Kapselung
- Manipulierbarkeit
- Konstruktor
- Objekt
- instanzspezifische und typspezifische Komp.

Abgeleiteter Typ

- Typbleitung und Erben
- Reimplementierung
- Polymorphismus

=> OOP lässt sich nicht in einem Satz erklären

© J F H Winkler, 2006

Nicht behandelt

this

Destruktor

Schachtelung von Objekttypen

generische Objekttypen

volle Palette der Exportstati

Mehrfachableitung

Interface

boxing / unboxing

Reflection

der Typ „class“

delegate

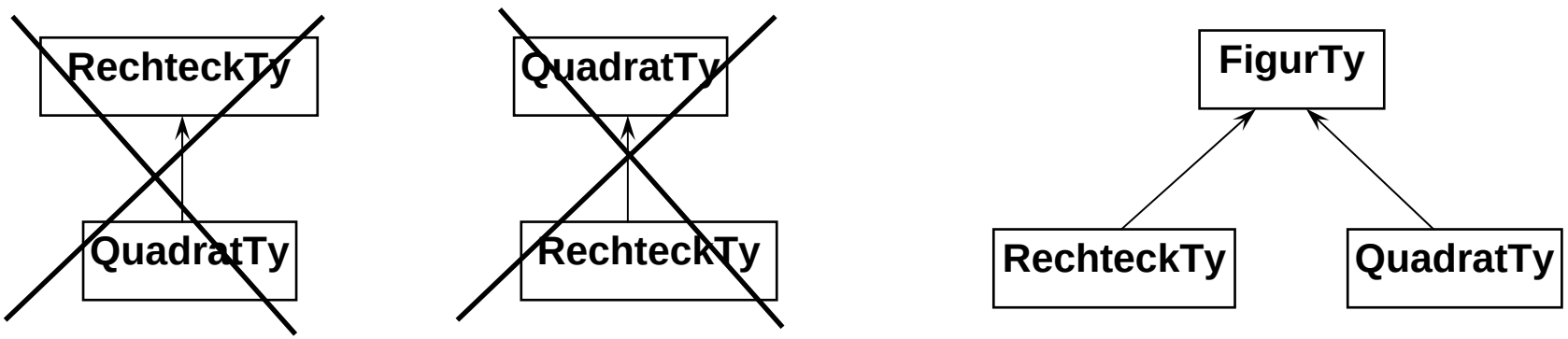
event

. . .

© J F H Winkler, 2006

Methodischer Hinweis

die Ableitungsrelation ist nicht isomorph zur
begrifflichen Klassifikation (Quadrat ist ein Rechteck mit …)



Winkler, Jürgen F.H.: Objectivism: “Class“ considered harmful.
Comm. ACM 35,8 (1992) 128..130

© J F H Winkler, 2006

Viel Erfolg beim Lehren der OOP

und

Vielen Dank fürs Zuhören

© J F H Winkler, 2006