
Objektorientierte Sprachen

Eine Sprache, die Objekte unterstützt, heißt objektbasiert

Eine klassenbasierte Sprache unterstützt zusätzlich Klassen

Eine objektorientierte Sprache unterstützt zusätzlich

- die Definition abstrakter Datentypen
- Vererbung
- Inclusion-Polymorphismus (polymorphe Variablen)
- dynamisches Binden von Funktionsaufrufen an Funktionen

Terminologie:

- Objekte sind Instanzen von Klassen
- Objekte enthalten Instanzvariablen, unterstützen Methoden
- Nachrichten werden an Objekte geschickt

©1998 Ghezzi and Jazayeri, from Programming Language Concepts, Third Edition, published by John Wiley.

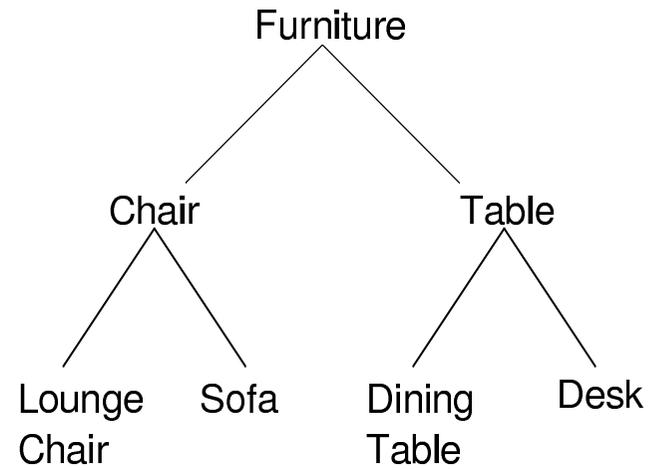


FIGURE 6.1 Sample inheritance hierarchy for furniture classes

Abgeleitete Klasse in C++

```
class stack {  
    public:  
        void push(int) {elements[top++] = i;}  
        int pop() {return elements[--top];}  
    private:  
        int elements[100];  
        int top = 0;  
};
```

```
class counting_stack: public stack {  
    public:  
        int size(); //return number of elements on the stack  
};
```

Polymorphismus, dynamisches Binden

```
stack s;                //nicht polymorph
counting_stack cs;
stack* sp = new stack(); //polymorph
counting_stack* csp = new counting_stack();
...
sp = csp;              //erlaubt
csp = sp;              //nicht erlaubt; statisch nicht zusicherbar
s = cs;                //erlaubt; wird zu Stack konvertiert
cs = s;                //nicht erlaubt; cs.size() undefiniert
...
sp->push(...);        //stack::push
csp->push(...);       //counting_stack::push
sp = csp;
sp->push(...);        //welches push?
```

Unterklassen versus Untertypen

Prinzip der Ersetzbarkeit: eine Instanz eines Untertyps ist überall erlaubt wo eine Instanz eines Obertyps erwartet wird.

Unterklassen sind Untertypen wenn

- nur Instanzvariablen und Methoden hinzugefügt werden;
- Methoden nur auf kompatible Weise umdefiniert werden.
Kompatibilität bedeutet, daß
 - die Methoden sich äquivalent verhalten
(nicht statisch prüfbar);
 - gleiche oder kompatible Signaturen haben
(statisch prüfbar, aber nicht in jedem Fall hinreichend).

Starke Typen und Polymorphismus

```
class base {...};  
class derived: public base {...};  
...  
base* b;  
derived *d;
```

Unter welchen Bedingungen können wir sicherstellen, dass eine Zuweisung $b = d$ nicht zu einem Typfehler führt?

`derived` ist ein Untertyp von `base` wenn `derived` überall verwendbar ist wo `base` erwartet wird, ohne dass diese Verwendung zu einem Typfehler führt.

Sind diese Bedingungen statisch prüfbar?

Überschreiben von Methoden

```
class polygon {
    public:
        polygon(...) {...} //constructor
        virtual float perimeter() {...};
        ...
}
class square: public polygon {
    public:
        square(...) {...} //constructor
        float perimeter() {...};
        ...
}
```

In C++, Java, Modula-3 muß die Signatur der überschreibenden Methode gleich jener der überschriebenen Methode sein.

Kontravariante Parametertypen

```
class base {
    public:
        virtual void fnc(s1 par) {...}
};
class derived: public base {
    public:
        void fnc(s2 par) {...}
};
base* b;
derived* d;
s1 v1;
s2 v2;
if(...) b = d;
b->fnc(v1); // was ist, wenn b vom Typ derived ist?
```

Kovariante Ergebnistypen

```
class base {
    public:
        virtual t1 fnc(...) {...}
};
class derived: public base {
    public:
        t2 fnc(...) {...}
};
base* b;
derived* d;
t0 v0;
if(...) b = d;
v0 = b->fnc(...); // was ist, wenn b vom Typ derived ist?
```

©1998 Ghezzi and Jazayeri, from Programming Language Concepts, Third Edition, published by John Wiley.

```
class point{
  public:
    x: float;
    y: float;
    bool equal (point p) //bool is defined as a boolean type
      {return (x == p.x && y == p.y);}
};
class colorPoint: public point{
  public:
    color: float;
    bool equal (colorPoint p) //bool is defined as a boolean type
      {return (x == p.x && y == p.y && color == p.colorPoint);}
};
```

FIGURE 6.2 Classes point and colorPoint

©1998 Ghezzi and Jazayeri, from Programming Language Concepts, Third Edition, published by John Wiley.

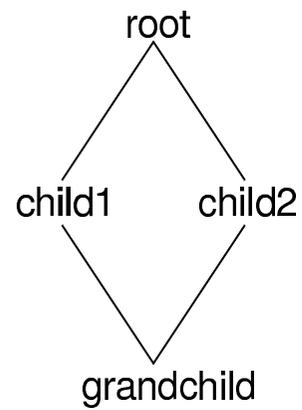


FIGURE 6.3 An example of diamond inheritance

C++

Konstruktoren initialisieren Objekte (Basisklasse zuerst).

Destruktoren räumen auf (Basisklasse zuletzt).

Automatische Objekte werden am Stack automatisch allokiert und deallokiert.

Objekte am Heap werden explizit mit `new` und `delete` allokiert und deallokiert; `new` und `delete` sind programmierbar.

Statische Variablen werden von allen Objekten einer Klasse geteilt (statische Allokation).

Zuweisungen und Gleichheitsvergleiche sind programmierbar (durch Überladen von `=` und `==`).

Virtuelle Funktionen in C++

```
class student {
    public:
        virtual void print() {...}
};
class college_student: public student {
    void print() {...}
};
student* s;
college_student* cs;
...
s->print(); //ruft student::print() auf
s = cs;    //erlaubt
s->print(); //ruft college_student::print() auf
```

©1998 Ghezzi and Jazayeri, from Programming Language Concepts, Third Edition, published by John Wiley.

```
class shape {
public:
    void draw() = 0;    // this and the others are pure virtual functions
    void move(...) = 0;
    void hide() = 0;
    point center;
};

class rectangle: public shape {
private:
    float length, width; // specific data for rectangle
public:
    void draw() {...};    // implementation for the derived pure virtual function
    void move(...) {...};
    void hide() {...};
};
```

FIGURE 6.4 A C++ abstract class using pure virtual functions

©1998 Ghezzi and Jazayeri, from Programming Language Concepts, Third Edition, published by John Wiley.

```
class stack{
    public:
        stack(); {top = 0;} //constructor
        void push(int) {s[top++] = i;};
        int pop() {return s[--top];};
    protected:
        int top;
    private:
        int s[100];
};

class counting_stack : public stack {
    public:
        int size(){return top;}; //return number of elements on the stack
};
```

FIGURE 6.5 Example of class inheritance (derivation) in C++

Arten des Polymorphismus in C++

Generizität (parametrischer Polymorphismus) durch Templates
— statisch aufgelöst

Inclusion-Polymorphismus durch (Mehrfach-)Vererbung
— statisch oder dynamisch (virtuelle Funktionen) aufgelöst

Überladen von Funktionen und Operatoren
— statisch aufgelöst

Explizite und implizite Typumwandlung, RTTI
— semantische Aktion zur Laufzeit

©1998 Ghezzi and Jazayeri, from Programming Language Concepts, Third Edition, published by John Wiley.

```
package Planar_Objects is  
  type Planar_Object is tagged  
  record  
    X: Float := 0.0; --default initial value of the center's x coordinate  
    Y: Float := 0.0; --default initial value of the center's y coordinate  
  end record;  
  function Distance (O: Planar_Object) return Float;  
  procedure Move (O: inout Planar_Object; X1, X2: Float);  
  procedure Draw (O: Planar_Object);  
end Planar_Objects;
```

FIGURE 6.6 An Ada 95 package that defines a tagged type Planar_Object

©1998 Ghezzi and Jazayeri, from Programming Language Concepts, Third Edition, published by John Wiley.

```
with Planar_Objects; use Planar_Objects;  
package Special_Planar_Shapes is  
  type Point is new Planar_Object with null record; --indicates no additions  
  procedure Draw (P: Point);  
  type Circle is new Planar_Object with  
    record  
      Radius: float;  
    end record;  
  procedure Draw (C: Circle);  
  type Rectangle is new Planar_Object with  
    record  
      Length, Width: Float; --sizes of edges of the rectangle  
    end record;  
  procedure Draw (T: Rectangle);  
end Special_Planar_Shapes;
```

FIGURE 6.7 Extending tagged types in Ada 95

Verwendung erweiterter Typen in Ada

O1: Planar_Object;

O2: Planar_Object := (1.0, 1.0);

C: Circle := (3.0, 4.5, 6.7);

...

O1 := Planar_Object(C);

C := (O2 **with** 4.7)

procedure Process_Shapes(O: Planar_Object'Class) **is**

...

begin

... Draw(O); ...

end Process_Shapes;

type Planar_Object_Ptr **is access** Planar_Object'Class;

©1998 Ghezzi and Jazayeri, from Programming Language Concepts, Third Edition, published by John Wiley.

```
package Planar_Objects is  
  type Planar_Object is abstract tagged null record;  
  function Distance (O: Planar_Object'Class) return Float is abstract;  
  procedure Move (O: inout Planar_Object'Class; X, Y: Float) is abstract;  
  procedure Draw (O: Planar_Object'Class) is abstract;  
end Objects;
```

FIGURE 6.8 An abstract type definition in Ada 95

Arten des Polymorphismus in Ada

Generizität, auch gebundene Generizität

— statisch aufgelöst

Inclusion-Polymorphismus durch (Einfach-)Vererbung

— statisch oder dynamisch (klassenweite Typen) aufgelöst

Überladen von Routinen und Operatoren

— statisch aufgelöst

Nur explizite Typumwandlung, Typvergleich

— semantische Aktion zur Laufzeit

Eiffel

class A feature

```
  fnc(t1: T1): T0 is ... do ... end -- fnc
end -- class A
```

class B inherit A redefine fnc feature

```
  fnc(s1: S1): S0 is ... do ... end -- fnc
end -- class B
```

```
  a: A;
```

```
  b: B;
```

```
  ...
```

```
  !!b;
```

```
  a := b;
```

```
  ... a.fnc(x) ...
```

Arten des Polymorphismus in Eiffel

Generizität, auch gebundene Generizität

— statisch aufgelöst

Inclusion-Polymorphismus durch (Mehrfach-)Vererbung

— dynamisch aufgelöst (expanded-Objekte statisch)

— Probleme bei der Typsicherheit

Überladen von Routinen und Operatoren

— statisch aufgelöst

überprüfter Zuweisungsversuch ($x \neq y$)

— semantische Aktion zur Laufzeit

Smalltalk

Objekte sind Instanzen von Klassen, die Instanz- bzw. Klassenvariablen und -methoden beschreiben.

Einfachvererbung; Methoden, die einer Nachricht entsprechen, werden dynamisch von unten nach oben gesucht.

Syntax: Objekt gefolgt von Nachricht und eventuell einem Parameter (z.B. `x > y`); bei mehreren Parametern:

```
lowerBound: low upperBound: high
```

```
T lowerBound: 6 upperBound: 95
```

Variablen sind untypisierte Referenzen → Polymorphismus

Objekte werden explizit erzeugt (z.B. `myPoint <- point new`);
Deallokation durch garbage collection

Smalltalk-Beispiele

```
[x <- 0. y <- 0] value
```

```
setZero <- [x <- 0. y <- 0]. ... setZero value
```

```
x > y
```

```
  ifTrue: [max <- x]
```

```
  ifFalse: [max <- y]
```

```
10 timesRepeat: [x <- x + a]
```

```
[x < b] whileTrue: [x <- x + a]
```

Eigenschaften von Java

Objekte werden am Heap allokiert, durch Referenzen adressiert, durch garbage collection deallokiert — keine Zeiger

Programme werden in portablen JVM-Code übersetzt
(Java ist als Netzwerk-Sprache geeignet)

Einfachvererbung von Klassen (ähnlich C++) und Mehrfachvererbung von Schnittstellen (ähnlich ML)

Klassen- (`static`) und Instanzvariablen bzw. -methoden

Trennung von Klassen und Modulen (Files und Directories)

Unterstützung von `final`-Klassen und -Methoden

Vordefinierte Klasse `Object` enthält `equals`, `clone`, etc.

©1998 Ghezzi and Jazayeri, from Programming Language Concepts, Third Edition, published by John Wiley.

```
class Planar_Objects {  
    protected float x, y;          //protected to be available to subclasses  
    public float distance ();      //return distance from origin  
    public void move (float x, y); //move object to point x,y  
    public void draw ();          //draw object on screen  
}
```

FIGURE 6.9 A Java class for Planar_Object

©1998 Ghezzi and Jazayeri, from Programming Language Concepts, Third Edition, published by John Wiley.

```
class Circle extends Planar_Object {
    private float Radius;
    public void Draw()          // draw a circle
        {...};
};
class Rectangle extends Planar_Object {
    private float Length, Width; // edge sizes of rectangle
    public void Draw()
        {...};
};
```

FIGURE 6.10 Extending a class in Java

©1998 Ghezzi and Jazayeri, from Programming Language Concepts, Third Edition, published by John Wiley.

```
interface Dictionary {  
    void Insert (String c; int i); //String type is defined by Java  
    int Lookup (String c);  
    void Remove (String c);  
}
```

FIGURE 6.11 Interface definition in Java for a dictionary

Verwendung von Schnittstellen in Java

```
class ArrayDict implements Dictionary {
    private String[] Names;
    private int[] Values;
    private int size = 0;
    public void Insert(String c; int i) {...}
    public int Lookup(String c) {...}
    public void Remove(String c) {...}
}
```

```
class ListDict1 implements Dictionary {...}
```

```
class ListDict2 extends SpList implements Dictionary {...}
```

```
interface PhoneList extends Dictionary, List {...}
```

©1998 Ghezzi and Jazayeri, from Programming Language Concepts, Third Edition, published by John Wiley.

```
class Consumer extends Thread {
    private Buffer buff;
    //construct a consumer
    Consumer (Buffer b) {
        buff = b;
    }
    public void run () {
        //remove item from buffer
        x = buff.get(); // buffer is an object shared with a producer
        //...do something with x...
    }
}
```

FIGURE 6.12 A consumer object in Java, inheriting from the Thread object

©1998 Ghezzi and Jazayeri, from Programming Language Concepts, Third Edition, published by John Wiley.

```
class Producer implements Runnable {
    private Buffer buff;
    //construct a producer object
    Producer (Buffer b) {
        buff = b;
    }
    public void run() {
        //produce things and put them in buffer
        buff.put (...);    //buff is a shared buffer object
        ...
    }
}
```

FIGURE 6.13 A producer object in Java, implementing the Runnable interface

Threads und Synchronisation in Java

```
Consumer C = new Consumer(aBuff);  
C.start();
```

```
new Consumer(aBuff).start();
```

```
public interface Runnable {  
    public void run();  
}
```

```
Producer P = new Producer(aBuff);  
Thread PT = new Thread(P);  
PT.start();
```

```

public class Buffer {
    private int n;           // size of buffer
    private int [] contents; // contents of buffer
    private int in, out = 0; // indexes of where to read from/write to
    private int total = 0;  // number of items in the buffer
    Buffer (int size) {
        n = size;
        contents = new int [n];
    }
    public synchronized void put (int item) {
        while (!(total < n))
            try { wait(); } // wait until there is space
            catch (InterruptedException e) { }
        contents [in] = item;
        System.out.println("Buffer: write at " + in + " item " + item);
        if (++in == n) in = 0;
        total++;
        notify(); // wake up any blocked threads
    }
    public synchronized int get () {
        int temp;
        while (!(total > 0))
            try { wait(); } // wait till there is something
            catch (InterruptedException e) { }
        temp = contents[out];
        System.out.println("Buffer: read from " + out + " item " + temp);
        if (++out == n) out = 0;
        total--;
        notify(); // wake up any blocked threads
        return temp;
    }
}

```

FIGURE 6.14 A synchronized buffer in Java

©1998 Ghezzi and Jazayeri, from Programming Language Concepts, Third Edition, published by John Wiley.

```
public class Main {  
    /** Entry point of the program */  
    public static void main (String args[])  
    {  
        Buffer B = new Buffer (100);    // create shared buffer  
        Consumer C = new Consumer (B); // create consumer  
        Producer P = new Thread (new Producer (B)); // create producer  
        C.start();                      // start consumer thread  
        P.start();                      // start producer thread  
    }  
}
```

FIGURE 6.15 Main program for the Java producer/consumer example

Arten des Polymorphismus in Java

Generizität, auch gebundene Generizität (seit Version 1.5)
— statisch aufgelöst

Inclusion-Polymorphismus durch (Mehrfach-)Vererbung
— dynamisch aufgelöst

Überladen von Methoden
— statisch aufgelöst

dynamisch überprüfte explizite Typumwandlung
— semantische Aktion zur Laufzeit