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

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?
```

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 Funtionen 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 = sc;      //erlaubt
s->print();  //ruft college_student::print() auf
```


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

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;

Arten des Polymorphismus in Ada

Generizität, teilweise auch gebundene Generizität
— statisch (oder auch dynamisch) 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 und dynamisch 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 ?= 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.

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 {...}
```


Threads und Synchronisation in Java

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

new Consumer(aBuff).start();

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

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

Arten des Polymorphismus in Java

Generizität nicht unterstützt

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

Überladen von Methoden
— statisch aufgelöst

überprüfte explizite Typumwandlung
— semantische Aktion zur Laufzeit