(Objektorientierte) Programmierkonzepte Vorlesung

Johannes Waldmann, HTWK Leipzig

20. März 2006

Einleitung

Inhalt

- (Objektorientierte) Konzepte zur Strukturierung von Programmen und Daten
- Modularisierung und Nachnutzung: Code einmal schreiben, oft benutzen
- Polymorphie (Vielgestaltigkeit): gleichen Code anwenden auf verschiedene Datentypen

Prinzip: *DRY* (don’t repeat yourself):

jedes Konzept soll genau eine, klar abgegrenzte Repräsentation im Programm (Quelltext) haben.

Arten der Polymorphie

- (statisches) Überladen (Java)

  ein Name ist mehrfach definiert, Auflösung zur Compilezeit

- Vererben/Überschreiben von Methoden ($=$ OO)

  (besser: Implementieren von Schnittstellen)

  Auflösung zur Laufzeit

- Generische Polymorphie (z. B. typsichere Operationen mit Containern)

  Auflösung zur Compilezeit

Polymorphie durch Schnittstellen

```
interface I { String foo (); }

class C implements I { .. String foo () { ..
class D implements I { .. String foo () { ..

class Top {
  void check (I x) {
    System.out.println (x.foo ());
  }
}
```

Die Methode `check` ist polymorph: zur *Laufzeit* ist das Argument `x` von irgendeinem Typ, der `I` implementiert.

Generische Polymorphie

```
interface List<E> { .. }
class LinkedList<E> implements List<E> { ..
  void add (E item) { .. }
  int size () { .. }
}
```

List und LinkedList sind *generische Typen*. Genauer: es sind *Funktionen*, die aus einem Typ einen anderen Typ herstellen.

die Methoden add und size sind generisch polymorph.

```
class Top {
  List<Integer> x = new LinkedList<Integer>(
  x.add (3); x.add (5); System.out.println (
}
```

Generische Polymorphie und Schnittstellen

Welches sollte der Typ einer Sortierfunktion sein?

```
static <E>
   List<E> sort (List <E> x); // fraglich
```

Beim Sortieren muß man Elemente vergleichen können, also muß man die Generizität von `sort` einschränken:

```
interface Comparable<E> {
   int compareTo (E item);
}
static <E implements Comparable<E>>
   List<E> sort (List <E> x);
```

Plan der Lehrveranstaltung

- Wiederholung Datentypen
- Typkonstruktoren, generisch polymorphe Funktionen
- Eingeschränkte Polymorphie: Typklassen ($=$ Interfaces in Java)
- generische Polymorphie in Java-1.5 (Beispiel: Collection Framework)
- . . . und in C++ (Standard Template Library) und/oder C#

Literatur

Software/Sprachen:

- Haskell `http://haskell.org/`
- Java-1.5 `http://java.sun.com/j2se/1.5.0/download.jsp`
- C++ `http://www.parashift.com/c++-faq-lite/`

(Lehr-)Bücher

- Andrew Koenig und Barbara Moo: *Intensivkurs C++*. Addison-Wesley/Pearson, 2003

Koordinaten

- Vorlesung: montags 17:15–18:45 HB207
- Übungen Z424:
  - Di gerade 15:30 und Fr ungerade 17:15
  - *oder* Do ungerade 9:30 und Fr gerade 11:15

Bewertung

zu jedem Thema (Haskell, Java, C++):

Programmieraufgaben und eine Kurzkontrolle (Mini-Klausur ca. 30 Minuten)

Datentypen

- Produkte
- Summen
- Potenzen (Exponentiale)

Produkte

Kreuzprodukte $T = A \times B$

C: struct, Pascal: record, Java: Object

Konstruktor: $A \times B \to T$; Zugriffe: $T \to A, T \to B$

Summen

(disjunkte) Vereinigungen (Summen) $T = A \cup B$

C: union, Pascal: record . . . case . . . of . . . end;

Falls $A, B, \ldots$ Einermengen: T ist Aufzählungstyp

Beispiel `enum boolean { false, true }`

Zugriff: Fallunterscheidung

„moderne“ Implementierung über Schnittstellen:

Interface repräsentiert disjunkte Summe aller Typen, die es implementieren

Potenzen (Exponentiale)

Exponentiale $T = A \to B = B^A$

Funktionen (andere Realisierungen: Arrays, Hashtabellen, Listen, Strings)

Konstruktion je nach Realisierung, Zugriff: Funktionsanwendung

Typ-Regel: wenn $f :: A \to B$ und $x :: A$, dann $f(x) :: B$

Bemerkung: $((A \to B) \wedge A) \to B$ ist allgemeingültige aussagenlogische Formel. Übereinstimmung ist kein Zufall!

Übung 11. Woche

Die Shell: Bash

Bash starten und Prompt-Symbol setzen:

```
bash -login
export PS1='\u@\h:\w\$ '
```

Zeileneditor: C-a, C-e, C-f, C-b, C-k, C-t

Geschichte: C-n, C-p

inkrementelles Suchen: C-r

diese Belegungen sind Standard, siehe GNU readline library `http://cnswww.cns.cwru.edu/~chet/readline/rltop.html`,

benutzt in bash, emacs, hugs ...

Der Haskell-Interpreter Hugs

`http://www.haskell.org/hugs/`

ist im Pool installiert. Pfade setzen (bash):

```
export PATH=/home/waldmann/built/bin:$PATH
export MANPATH=/home/waldmann/built/man:$MAN
export LD_LIBRARY_PATH=/home/waldmann/built/
```

(das kann man alles in `~/.bashrc` schreiben)

Beispiel: auf wieviele Nullen endet 100 Fakultät?

```
hugs +tT

[ 1 .. 100 ]
product [ 1 .. 100 ]
show $ product [ 1 .. 100 ]
reverse $ show $ product [ 1 .. 100 ]
```

```
takeWhile ( == '0' ) $ reverse $ show $ prod
length $  takeWhile ( == '0' ) $ reverse $ s
```

Polymorphe Typen, Funktionen

Welche Typen haben diese Funktionen?

```
reverse
length
( == '0' )
takeWhile
```

Rufen Sie diese Funktionen auf (d. h.: bilden Sie typkorrekte Ausdrücke, die diese Funktionen benutzen)!

Betrachten Sie weitere Funktionen auf Listen! Siehe Modul `List` in Standardbibliothek `http://www.haskell.org/onlinereport/` bzw. `http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-List.html`

Funktionen in Haskell

Mehrstellige Funktionen

```
drop 3 "foobar" = "bar"
drop :: Int -> [a] -> [a]
```

Wenn $f :: A_1 \to A_2 \to \ldots A_k \to B$,
dann ist f eine k-stellige Funktion
mit Argumenttypen $A_1, \ldots, A_k$
und Ergebnistyp B.

Beachte: das ist eine einstellige Funktion:

```
drops (3, "foobar") = "bar"
drops :: (Int, [a]) -> [a]
```

Operatoren und Funktionen

Operator: zwischen Argumenten

Funktion: vor Argumenten.

die folgenden Schreibweisen sind äquivalent:

```
x == y        und        (==) x y
x + y         und        (+) x y
```

Spezielle Operator-Syntax (Sections)

```
A  -> '('   Op   ')' -- zweistellige Fkt.

    | '('   Op A ')'
       -- left  section, einst. Fkt

    | '(' A Op   ')'
       -- right section, einst. Fkt
```

Beispiele: alle folgenden Ausdrücke sind äquivalent

```
"foo" ++ "bar",  (++) "foo" "bar",
("foo" ++) "bar", (++ "bar") "foo"
```

Anwendungen

```
filter odd      [1,2,3,4,5,6,7] = [1,3,5,7]
filter ( < 4 ) [1,2,3,4,5,6,7] = [1,2,3]
```

Hierbei ist `( < 4 )` ein *Ausdruck*, der die Funktion `\ x -> x < 4` beschreibt.

```
zipWith (*) [1,2] [4,5,6] = [4,10]
```

hier beschreibt der *Ausdruck* `(*)` die Funktion `\ x y -> x * y`.

Typ von `zipWith`?

Anonyme Funktionen

(Funktionen ohne Namen)

mathematische Notation:

Lambda-Kalkül, Alonzo Church, 1936

Beispiel: $\lambda x.2 \cdot x + 3$ `\ x -> 2 * x + 3`

Die Deklarationen

```
f x = 2 * x + 3
```

und

```
f = \ x -> 2 * x + 3
```

sind äquivalent.

Funktionen höherer Ordnung

```
odd :: Int -> Boolean
odd 1 = True ; odd 5 = True ; odd 2 = False

filter odd [1,5,2,7,4,5,9] = [1,5,7,5,9]
```

Typ von `filter`?

```
filter :: (a -> Bool) -> [a] -> [a]
```

ist zweistellige Funktion, erstes Argument ist selbst eine Funktion.

```
partition odd [1,5,2,7,4,5,9]
  = ( [1,5,7,5,9], [2,4] )
```

Typ von `partition`?

Partielle Anwendung von Funktionen

- mehrstellige Funktionen kann man *partiell* anwenden
- ($=$ einige, aber nicht alle Argumente angeben).
- Das Resultat ist wieder eine Funktion.
- Beispiel: `partition odd`

Datentypen in Haskell

Algebraische Datentypen

disjunkte Vereinigung (erkennbar am Konstruktor)
von Kreuzprodukten mit benannten Komponenten.

```
data Tree a = Leaf
            | Node { key   :: a
                   , left  :: Tree a
                   , right :: Tree a
                   }

t :: Tree Int
t = Node
  { key = 5
  , left = Leaf
  , right = Node { key = 7, left = Leaf, right = Leaf
```

```
}
```

Algebren, Signaturen

- Eine (mehrsortige) *Signatur* Σ

  ist eine Menge von Funktionssymbolen, jeweils mit Typ.

  im OO-Entwurf ist das eine Schnittstelle (Interface)

- Eine Σ-Algebra A

  ist eine Zuordnung von Sorten zu Mengen und Symbolen zu Funktionen (mit dem richtigen Typ).

  im OO-Entwurf ist das eine Implementierung (Klasse)

- Eine wichtige Σ-Algebra ist die *Term-Algebra*:

  die Mengen sind gerichtete, geordnete, markierte Bäume, die Funktionen setzen Bäume zusammen.

Vordefinierte Datentypen

bereits (so ähnlich) eingebaut:

```
data Bool = False |  True
data  [a]   = []
            | (:) { head :: a
                  , tail :: [a]
                  }

1 : (2 : (3 : (4 : [])))
  = [1, 2, 3, 4] :: [ Int ]
```

Für `xs /= []` gilt: `head xs : tail xs = xs`

Fallunterscheidungen (case ... of)

```
size :: Tree a -> Int
size t = case t of
    Leaf      -> 1
    Node { } -> size (left  t)
              + size (right t)
```

passend zu Data-Deklaration

```
data Tree a = Leaf
            | Node { key   :: a
                   , left  :: Tree a
                   , right :: Tree a
                   }
```

Varianten von Fallunterscheidungen

benutzt *Zugriffsfunktion* (`tail`):

```
length :: [a] -> Int
length l = case l of
    [] -> 0
    _  -> 1 + length (tail l)
```

Variablenbindung (für `x`, `xs`) durch *pattern matching*:

```
length l = case l of
    []     -> 0
    x : xs -> 1 + length xs
```

desgl. in Deklarationen mit mehreren Klauseln:

```
length []       = 0
length (x : xs) = 1 + length xs
```

Suchbäume: Herstellen

aus (ungeordneter) Liste herstellen:

```
import Data.List (partition)

suchbaum :: [ Int ] -> Tree Int
suchbaum []       = Leaf
suchbaum (x : xs) =
   let ( low, high ) = partition ( < x ) xs
   in  Node { key   = x
            , left  = suchbaum low
            , right = suchbaum high
            }
```

Syntax: Lokale Deklarationen: let . . . in . . .

A: Ausdruck, d: Name, T: Typ, W, A: Ausdruck

```
    let d :: T
        d = W
        ..
    in  A
```

vergleichbar mit (Java):

```
{ T d = W; // Deklaration mit Initialisierun
; return A
}
```

Suchbäume: Durchlaufen

Inorder-Durchquerung der Knoten:

```
inorder :: Tree a -> [a]
inorder t = case t of
    Leaf       -> []
    Node { key = k, left = l, right = r } ->
       inorder l ++ [ k ] ++ inorder r
```

Sortieren:

```
sort :: [ Int ] -> [ Int ]
sort xs = inorder ( suchbaum xs )
```

variablenfreie Schreibweise durch Komposition von Funktionen:

```
sort = inorder . suchbaum
```

Aufgabe: welche Typ hat `(.)` ?

Übung 12. Woche

Emacs . . . kann alles außer Tee kochen

```
$ emacs -font *20 &
```

File öffnen (in neuem Buffer): C-x C-f, Wechsel zwischen Buffern: C-x b, alle Buffer speichern: C-x s, Editor verlassen: C-x C-c.

Kommando abbrechen: C-g (*nicht C-c*!), zurücknehmen: C-x u (undo), Hilfe: C-h i (info - beenden mit q).

Textblöcke: zwischen *mark* (setzen mit C-space) und *point* (= Kursor).

cut (block): C-w (wipe), (zeile): C-k (kill), *paste*: C-y (yank)

Rechtecke: C-x r k (kill-rectangle), C-x r y (yank-rectangle)

`M-x` („Meta-x“) = (ESC, dann x) = (ALT *und* x)

`M-x gomoku`, `M-x dunnet`

Emacs/Hugs als Haskell-IDE

- Emacs für Quelltext im Hintergrund starten:
  `emacs Seminar.hs &`

- Hugs im Vordergrund und Quelltext laden
  `hugs +Tt Seminar.hs`

- – Editieren,
  – Emacs:Speichern (C-x C-s),
  – Hugs: Re-Laden (:r)

Suchbäume

- deklariere Datentyp für binäre Bäume mit Schlüsseln vom Typ a:

```
-- brauchen wir später:
import List (partition)

data Tree a = Leaf
            | Node { key :: a
                   , left :: Tree a
                   , right :: Tree a
                   }
    deriving Show
        -- entspr. automatischer Deklaratic
```

```
        -- der Java-Methode toString

t :: Tree Int
t = Node { key = 5
         , left = Node { key = 3, left = Le
         , right = Leaf
         }
```

- Zähle Blätter in einem Baum:

```
leaves :: Tree a -> Int
leaves t = case t of
    Leaf ->  1
    Node { key = k, left = l, right = r } -
         leaves l + leaves r
```

- definiere Funktion zum Herstellen eines Suchbaumes aus einer Liste:

```
suchbaum :: [ Int ] -> Tree Int
suchbaum [] = Leaf
suchbaum (x : xs) =
    let ( smaller, larger ) = partition ( <
    in  Node { key = x
             , left = suchbaum smaller
             , right = suchbaum larger
             }
```

  testen z. B. mit `suchbaum [8,1,4,3,9]`

- definiere Funktion zur Herstellen der Inorder-Reihenfolge der Schlüssel

```
inorder :: Tree a -> [a]
inorder t = case t of
    Leaf -> ???
    Node { key = k, left = l, right = r } -
        inorder l ++ [ k ] ++ inorder r
```

testen z. B. mit `inorder $ suchbaum [8,1,4,3,9]`

- definiere Funktion zum Sortieren:

```
sort :: [ Int ] -> [ Int ]
sort xs = inorder $ suchbaum xs
```

Welche Komplexität hat dieser Algorithmus?

Suche in Suchbäumen

Implementieren Sie eine Funktion

`contains :: Tree Int -> Int -> Bool`

so daß `contains t x` genau dann wahr ist, wenn `x :: Int` als Schlüssel in `t :: Tree Int` vorkommt.

Achten Sie auf

- Typkorrektheit
- semantische Korrektheit
- Effizienz

Übungen: append

implementiere

```
append :: [a] -> [a] -> [a]
-- append "foo" "bar" = "foobar"
```

benutze Fallunterscheidung nach dem ersten Argument:

```
append []       ys = ???
append (x : xs) ys = ???
```

Diese Funktion ist schon vordefiniert als Operator ++

Vorlesung 13. Woche

Muster

```
leaves :: Tree a -> Int
leaves t = case t of
    Leaf ->  1
    Node { key = k, left = l, right = r }
         -> leaves l + leaves r
inorder :: Tree a -> [a]
inorder t = case t of
    Leaf -> []
    Node { key = k, left = l, right = r }
         -> inorder l ++ [ k ] ++ inorder r
```

Muster

```
f :: Tree a -> b
f  t = case t of
     Leaf -> c
     Node { left = l, key = k, right = r }
          -> h (f l) k (f r)
```

jeder Konstruktor wird durch eine Funktion ersetzt: der nullstellige Konstruktor `Leaf` durch eine nullstellige Funktion (Konstante) `c`, der dreistellige Konstruktor `Node` durch eine dreistellige Funktion `h`

Für `c = 1 , h x k y = x + y` erhalten wir `leaves`,

für `c = [], h x k y = x ++ [k] ++ y` erhalten wir `inorder`.

Muster als Funktionen höherer Ordnung

```
tfold :: b
      -> ( b -> a -> b -> b )
      -> Tree a
      -> b
tfold c h t = case t of
    Leaf -> c
    Node { left = l,  key = k, right = r }
        -> h (tfold c h l) k (tfold c h r)
```

Dann ist

```
leaves’  =
  tfold ( \ fl k fr -> fl          + fr )  1
inorder’ =
  tfold ( \ fl k fr -> fl ++ [k] ++ fr ) []
```

Zusammenfassung Muster

- zu jedem Typ-Konstruktor `T` gehört ein fold-Muster, das „natürliche“ Funktionen `T a -> b` beschreibt.

  Das Muster ist eine Funktion höherer Ordnung, ihre Argumente sind Funktionen—für jeden Konstruktor eine.

  (Wie sieht also das fold-Muster für Listen aus?)

- wenn die Programmiersprache es gestattet, Funktionen höherer Ordnung zu benutzen, dann sind Entwurfsmuster (bes. Verhaltensmuster) einfach selbst Funktionen.

  . . . wenn nicht, muß man diese erst „erfinden“ und dann dicke Bücher darüber schreiben.

Funktionales Programmieren in Java

in Java gibt es keine Funktionen höherer Ordnung (Methoden können nicht Argument oder Resultat von Methoden sein).

wenn man diese aber (durch ordentliches Design) doch braucht, muß man sie simulieren, vgl. ActionListener:

gemeint ist: führe bei jeder Nutzeraktion an Komponente k die Aktion (Prozedur, Methode) m aus.

geschrieben wird: eine (lokale oder anonyme) Klasse M, die m als einzige Methode hat, und übergeben an k wird eine Instanz (ein Objekt) dieser Klasse M.

Übung 13. Woche

Vorbereitung

- Deklaration `data Tree a = ...`, `tfold :: ...` und `suchbaum :: ...` aus voriger Vorlesung/Übung kopieren.
- was ist der allgemeinste Typ von `suchbaum`?
- Funktionen `leaves, inorder` durch `tfold` definieren und ausprobieren

Fold für Listen

- Definieren Sie das Rekursionmuster:

```
lfold :: b -> (a -> b -> b) -> [a] -> b
lfold c h l = case l of
    []     -> undefined
    x : xs -> undefined
```

- Definieren Sie damit Funktionen:
  - Summe aller Elemente einer Liste,
  - Produkt aller Elemente einer Liste,
  - Länge einer Liste

Aufgaben

bis nächste Woche, Lösungen (alle in einer Datei) über Autotool einsenden

- definiere Funktionen `nodes` (Anzahl aller *inneren* Knoten) durch `tfold`
- was tut diese Funktion:

```
foo = tfold Leaf ( \ fl k fr -> Node { key
```

- definiere Funktion ähnlich `contains`, die Schlüssel in einem Suchbaum wiederfindet. Benutze `tfold`.

```
contains :: Ord a => a -> Tree a -> Bool
-- contains 2 $ suchbaum [ 5,2,6,4,7,4,9 ]
```

```
contains x = tfold False ( \ cl k cr -> unc
```

Polymorphie/Typklassen

Einleitung

```
reverse [1,2,3,4] = [4,3,2,1]
reverse "foobar"  = "raboof"
reverse :: [a] -> [a]
```

`reverse` ist polymorph

```
sort [5,1,4,3] = [1,3,4,5]
sort "foobar"  = "abfoor"


sort :: [a] -> [a] -- ??
sort [sin,cos,log]  = ??
```

`sort` ist *eingeschränkt polymorph*

Der Typ von sort

zur Erinnerung: `sort` enthält:

```
let ( low, high ) = partition ( < ) xs in ..
```

Für alle `a`, die für die es eine Vergleichs-Funktion gibt, hat `sort` den Typ `[a] -> [a]`.

```
sort :: Ord a => [a] -> [a]
```

Hier ist `Ord` eine *Typklasse*, so definiert:

```
class Ord a where
    compare :: a -> a -> Ordering
data Ordering = LT | EQ | GT
```

vgl. Java:

```
interface Comparable<T>
    { int compareTo (T o); }
```

Instanzen

Typen können Instanzen von *Typklassen* sein.

(OO-Sprech: Klassen implementieren Interfaces)

Für vordefinierte Typen sind auch die meisten sinnvollen Instanzen vordefiniert

```
instance Ord Int ; instance Ord Char ; ...
```

weiter Instanzen kann man selbst deklarieren:

```
data Student = Student { vorname  :: String
                       , nachname :: String
                       , matrikel :: Int
                       }
instance Ord Student where
    s < t  =  matrikel s < matrikel t
```

Typen und Typklassen

In Haskell sind diese drei Dinge *unabhängig*

1. Deklaration einer Typklasse (= Deklaration von abstrakten Methoden)
   `class C where { m :: ... }`
2. Deklaration eines Typs (= Sammlung von Konstruktoren und konkreten Methoden) `data T = ...`
3. Instanz-Deklaration (= Implementierung der abstrakten Methoden) `instance C T where { m = ... }`

In Java sind 2 und 3 nur *gemeinsam* möglich

`class T implements C { ... }`

Das ist an einigen Stellen nachteilig und erfordert Bastelei: wenn `class T implements Comparable<T>`, aber man die T-Objekte anders vergleichen will?

Man kann deswegen oft die gewünschte Vergleichsfunktion separat an Sortier-Prozeduren übergeben.

...natürlich nicht die Funktion selbst, Java ist ja nicht funktional, sondern ihre Verpackung als Methode eines Objekts einer Klasse, die

```
interface Comparator<T>
   { int compare(T o1, T o2); }
```

implementiert.

Klassen-Hierarchien

Typklassen können in Beziehung stehen.

Ord ist tatsächlich abgeleitet von Eq:

```
class Eq a where
   (==) :: a -> a -> Bool
class Eq a => Ord a where
   (<)  :: a -> a -> Bool
```

also muß man erst die Eq-Instanz deklarieren, dann die Ord-Instanz.

Beachte: das sind Abhängigkeiten (Ableitungen, Vererbungen) zwischen Typklassen (Interfaces) — *gut*, . . . hingegen sind Abhängigkeiten (Vererbungen) zwischen Implementierungen *schlecht* (und in Haskell gar nicht möglich. . .)

Die Klasse Show

```
class Show a where
    show :: a -> String
```

vgl. Java: toString()

Der Interpreter Hugs gibt bei Eingab `exp` (normalerweise) `show exp` aus.

Man sollte (u. a. deswegen) für jeden selbst deklarierten Datentyp eine Show-Instanz schreiben.

...oder schreiben lassen: `deriving Show`

Automatisches Ableiten von Instanzen (I)

```
data Tree a = Node { key :: a
                   , left :: Tree a
                   , right :: Tree a
                   }
            | Leaf
instance Show a => Show (Tree a) where
    show t @ (Node {}) =
        "Node{" ++ "key="  ++ show (key  t) ++ ","
                ++ "left=" ++ show (left t) ++ ","
                ++ "left=" ++ show (left t) ++ "}"
    show Leaf = "Leaf"
```

Das kann der Compiler selbst:

```
data Tree a = ... deriving Show
```

Generische Instanzen (I)

```
class Eq a where
    (==) :: a -> a -> Bool
```

Lexikografische Ordnung auf Listen:

wenn a in Eq, *dann* [a] in Eq:

```
instance Eq a => Eq [a] where
    []       == []
        =  True
    (x : xs) == (y : ys)
        =  (x == y) && ( xs == ys )
    _        == _
        =  False
```

Generische Instanzen (II)

```
class Show a where
    show ::  a -> String

instance Show a => Show [a] where
    show [] = "[]"
    show xs = brackets
            $ concat
            $ intersperse ","
            $ map show xs

show 1 = "1"
show [1,2,3] = "[1,2,3]"
```

Überlappende Instanzen

Wegen `String = [Char]` gilt nach bisheriger Deklaration:

```
show ’f’ = "’f’"
show "foo" = "[’f’,’o’,’o’]"
```

Erwünscht ist aber:

```
instance Show String where
    show cs = "\"" ++ cs ++ "\""
show "foo" = "\"foo\""
```

Diese Instanz-Deklaration überlappt mit generischer.

Möglicher Ausweg: die speziellere Instanz gewinnt, also hier: `instance Show [Char]` gegen `instance Show [a]`.

Typklassen als Prädikate

Man unterscheide *gründlich* zwischen Typen und Typklassen (OO: zwischen Klassen und Schnittstellen).

Eine Typklasse C ist ein (einstelliges) *Prädikat* auf Typen T: Die Aussagen $C(T_1), C(T_2), \ldots$ sind wahr oder falsch.

Auch mehrstellige Prädikate (Typklassen) sind möglich und sinnvoll. (Haskell: multi parameter type classes, Java: ?)

Übung 14. Woche

Aufgaben zu Typklassen

Deklarieren Sie

```
data Buch = Buch
          { autor :: String
          , titel :: String
          , ort   :: String
          , jahr  :: Int
          }
    deriving ( Eq, Ord )
b3 :: Buch
b3 = Buch { autor = "Donald E. Knuth"
```

```
          , titel = "The Art Of Computer Pro
          , ort   = "Reading, Mass."
          , jahr  = 1998
          }
```

und implementieren Sie

```
instance Show Buch where
    show b =  ...
```

Deklarieren Sie noch ein Buch `b2` (suchen Sie Informationen zu ISBN 0-262-03293-7) und werten Sie `b2 < b3` aus. Welche Implementierung von `(<)` wurde durch `deriving Ord` generiert? Ändern Sie in der Deklaration des Typs `Buch` die Reihenfolge der Komponenten. Wie wirkt sich das auf die generierte Version von `(<)` aus?

Typklassen (Erweiterungen)

Beispiel: Klassen für Brettspiele

```
class Brett b where
    dran   :: b -> Farbe
    gewonnen :: b -> Maybe Farbe
class Brett b => Zug b z where
    next :: b -> z -> Maybe b
        data Maybe b = Nothing | Just b
class Shaped b s where
    shape  :: b -> s
    shaped :: s -> b
replay :: ( Brett b, Zug b z , Shaped b s )
```

```
=> s -> [z] -> b
```

Brettspiele: Havannah

```
data Hexagon = Hexagon { breit :: Int }
data Havannah = Havannah
    { h_shape :: Hexagon
    , stones :: Array Farbe ( Set Point )
    , ist_dran :: Farbe
    }
instance Shaped Havannah Hexagon where
    shaped h = Havannah { h_shape = h
        , stones = array (Schwarz, Weiss)
                [ ( Schwarz, emptySet ), ( Weiss, empt
        , ist_dran = Schwarz
        }
data Satz = Put Point | Pass | Resign
instanz Zug Havannah Satz where
    erlaubt s z = ...
    next s z = if erlaubt s z then case z of (Put p)
```

Havannah - Online

- offizielle Havannah-Seite vom Erfinder Christian Freeling: `http://www.mindsports.net/index-mindsports.html`

  (1000 EUR Preisgeld für Programm, das ihn schlägt)

- lokale Startseite (inkl. Applet zum Selbst-Spielen): `http://dfa.imn.htwk-leipzig.de/havannah/`

- Quelltext-Archiv (Server und Bot in Haskell, Applet in Java): `http://dfa.imn.htwk-leipzig.de/cgi-bin/cvsweb/havannah/?cvsroot=havannah`

- Spieltheorie-Vorlesung:

http://www.imn.htwk-leipzig.de/~waldmann/edu/ancient/ws01/kst/

Multi-Parameter-Klassen

Eine Typklasse (Interface) ist ein einstelliges Prädikat. ein Typ erfüllt es (ist Instanz, implementiert es), oder nicht.

```
class Ord a where ... ; instance Ord Student where ..
```

Oft benötigt man mehrstellige Prädikate (Relationen)

```
class Brett b => Zug b z where ...
instance Zug Havannah Satz where ...
```

diese werden von *Tupeln* von Typen erfüllt (oder nicht).

(geht das in „klassischen“ OO-Sprachen?)

Man kann zusichern, daß die Relation eine Funktion ist (*functional dependency*):

```
class Problem p i b | (p, i) -> b
```

Automatisches Ableiten (II)

`deriving` geht nur für vordefinierte Klassen.

für nutzerdefinierte Klassen müßte der Nutzer gewünschte Ableitungsregeln selbst angeben.

das heißt dann *polytypic programming*, `http://www.cs.chalmers.se/~patrikj/poly/`

unterstützt z. B. durch Präprozessor `DrIFT`

oder (wegen typischer Nachteile von Präprozessoren) durch Sprach/Compiler-Erweiterungen: `http://www.cs.uu.nl/research/projects/generic-haskell/`

Testat 1

http://www.imn.htwk-leipzig.de/~waldmann/edu/ss05/oo/punkte/testat_haskell.text

Einführung Java-1.5

Neuheiten in Java-1.5

`http://java.sun.com/j2se/1.5.0/lang.html`

- Java-1.5 ist tot, es lebe Java-5.0
- generics ($\Rightarrow$ Typsicherheit bei Collections)
- enhanced `for` loop,
- autoboxing,
- typesafe enumerations,
- (static imports, annotations, varargs)

Java-Wiederholung: Sprache

Rechtschreibung, Grammatik:

- Ausdrücke (atomar, zusammengesetzt)
- Anweisungen (atomar, zusammengesetzt)
- Deklarationen
- Export-Modifier (public, protected, default, private)

Java-Wiederholung: Ausdrucksformen

- Anweisungen: benutze Einrückungen (Block-Schachteltiefe), nur eine Anweisung pro Zeile, nach *if* immer Block
- Deklarationen: sinnvolle Namen, Pakete klein, Klassen groß, Methoden und Attribute klein, Attribute mit Unterstrich (`_foo`)
- Deklarationen immer
  - so *spät*, so *lokal* und so *final* wie möglich
- Variablen immer initialisieren

Java-Wiederholung: Objekte

- Objekte, Klassen, Pakete
- Attribute (Variablen), Methoden
- Modifier (static, default)
- abstrakte Methoden, Klassen
- Überschreiben und Überladen von Methoden

Java-Wiederholung: Umgebung

- Quelltext der Klasse `Bar` aus Paket `foo` steht in `foo/Bar.java`
- Durch Kompilation `javac foo/Bar.java` (Dateiname!) entsteht daraus Class-File `foo/Bar.class`
- mehrere Class-Files kann man in einem Archiv (jar) zusammenfassen

- Class-File kann man ausführen durch
  `java foo.Bar arg0 arg1 ...` (Klassenname!),

  falls es eine Methode
  `public static void main (String [] argv)`
  enthält.

- falls `class Foo extends Applet`, und `bar.html` enthält

```
<APPLET CODE="Foo.class">
    <PARAM NAME="beet" VALUE="hoven">
</APPLET>
```

  dann `appletviewer bar.html`

Java-1.5 benutzen

- (zuhause) selbst installieren: `http://java.sun.com/j2se/1.5.0/download.jsp`
- (Sun-Pool) benutzen: in `.bashrc` schreiben:

```
export PATH=/usr/local/j2sdk1.5.0/bin:$PATH
```

- compilieren:
  `javac -source 1.5 -Xlint Foo.java`
- ausführen: `java Foo`, `appletviewer Foo.html`

oder Eclipse-3.1M5 (Sun-Pool: ja, PC-Pool: noch nicht?)
Window → Preferences → Java → Compiler → Compiler Compliance Level → 5.0

Kompatibilität zwischen 1.5 und 1.4

- 1.4-Quellen lassen sich auch in 1.5 übersetzen (mit Warnungen wg. Typ-Unsicherheiten bei Collections)
- die 1.5-Class-Files laufen auch auf 1.4-JVMs? (wurde anfänglich behauptet, dann aber stillschweigend aufgegeben)
- (Die Generics sind daran nicht schuld—der Witz besteht ja gerade darin, daß nur zur Compilezeit schärfer geprüft wird, damit man das zur Laufzeit weglassen kann.)
- Ausweg: 1.5-Class-Files nach 1.4 rückportieren mit `http://retroweaver.sourceforge.net/`

Verbesserte For-Schleifen

(einfachster Fall: über Elemente eines Arrays)

```
int a [] = { 2, 7, 1, 8, 2, 8 };
int sum = 0;
```

bisher:

```
for (int i=0; i<a.length; i++) {
    sum += a[i];
}
```

jetzt:

```
for (int x : a) {
    sum += x;
}
```

allgemein: `for (Typ name : Collection) {}`

Generische Klassen und Methoden

wesentliches Ziel:

(compile-Time-)Typsicherheit bei polymorphen Collections.

Gilad Bracha: *Generics in the Java Programming Language* (Tutorial), `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`

Java(TM) 2 Platform, Standard Edition, v 1.5.0 API Specification: `http://java.sun.com/j2se/1.5.0/docs/api/index.html`

Listen

Aufgabe: eine Liste `[1, 2 .. n ]` erzeugen.

```
static List<Integer> make (int n) {
    List<Integer> l =
        new LinkedList<Integer> ();
    for (int i = 1; i <= n; i++) {
        l.add (i);
    }
    return l;
}
```

Hierbei sind

- `List` ein Interface (abstrakter Datentyp)
- `LinkedList` eine Implementierung (konkreter Datentyp)

Auto-Boxing

Java unterscheidet:

- elementare Typen (int, char, boolean,...)
- Objekttypen

elementare Typen gehören zu keiner Klasse $\rightarrow$ besitzen keine Methoden.
Das ist unpraktisch, deswegen gibt es Wrapper-Klassen (Integer, Character, Boolean,...):

- boxing: `Integer b = new Integer (5);`
- unboxing: `int i = b.intValue ();`

Das ist immer noch unpraktisch, $\rightarrow$ boxing und unboxing in Java-1.5 automatisch.

For-Loop für Listen

```
static int product (List<Integer> c) {
  int p = 1;
  for (int x : c) {
    p *= x;
  }
  return p;
}
```

in Übungen: Beispiel aus Vorlesung nachrechnen, Listen-Operationen erkunden, Listen von Listen von Zahlen bauen usw.

Übung 15. KW

```
export PATH=/home/waldmann/built/bin:$PATH
eclipse &
```

Window $\rightarrow$ Preferences $\rightarrow$ Java $\rightarrow$ Compiler $\rightarrow$ compiler compliance level $\rightarrow$ 5.0

Window $\rightarrow$ Preferences $\rightarrow$ Java $\rightarrow$ Compiler $\rightarrow$ Errors/Warning $\rightarrow$ JDK 5.0 Options $\rightarrow$ Unchecked type ops $\rightarrow$ Error (anderes $\rightarrow$ Warning)

Beispiele aus Vorlesung ausprobieren:

- for-loop über Array
- Liste $[1 \ldots n]$ erzeugen, alle Elemente multiplizieren.

implementiere polymorphe Funktion

```
static <E> LinkedList<E> shuffle (List<E> in
```

soll aus $[1, 2, 3, 4, 5, 6, 7, 8]$ die Liste $[7, 5, 3, 1, 2, 4, 6, 8]$ erzeugen, usw.

Benutze zum Zugriff auf `in` *nur*

`for (E x : in) { ... }` (d. h. *kein* `in.get()`)

Benutze zum Erzeugen der Ausgabeliste

```
LinkedList<E> out = new LinkedList<E> ();
```

die Methoden `addFirst, addLast` sowie eine boolesche Variable zum Umschalten zwischen beiden.

Rufen Sie `shuffle` mehrfach auf, z. B.

```
public static void main(String[] argv) {
   List<String> in = Arrays.asList (argv);
   // List<Integer> = make (8);
   System.out.println (in);
   in = shuffle (in);
```

```
    System.out.println (in);
}
```

(Fügen Sie eine Schleife ein.)

Übung KW 16

Thema: binäre (Such)bäume, vergleiche

```
data Tree a = Leaf
            | Node { left :: Tree a, key ::
```

Aufgaben:

- richtiges Design der Klassen
- einen Test-Baum (vollständiger binärer Baum der Tiefe n) herstellen (Schlüsseleinträge beliebig)
- einen Baum ausgeben (`toString`)
- ein Objekt suchen (`contains`)
- ein Objekt einfügen (`add`)

Einzelheiten:

- nach außen sichtbar:

```
package data;
public class Tree<E extends Comparable<E>>
private Entry<E> root = null;
        ...
}
```

- Implementierung:

```
package data;
public class Entry<E> {
Entry<E> left;
E key;
```

```
Entry<E> right;
        ...
}
```

Eclipse → Source → generate constructor using fields

- Beabsichtige Benutzung:

```
package data;
public class TreeTest {
public static void main(String[] args) {
Tree<Integer> t = Tree.full (2);
System.out.println (t);
}
}
```

soll das ausgeben:

```
Node{left=Node{left=Leaf,key=1,right=Leaf},
```

- implementiere die dazu benötigen Methoden `full` und `toString`.

Java-Collections (V 26. 4.)

Collections — Überblick

- `interface Collection<E>` Gruppe von Elementen, evtl. mit Duplikaten, evtl. geordnet, evtl. indiziert
  - `interface List<E>`
    Duplikate erlaubt, Zugriff über Index
  - `interface Set<E>`
    keine Duplikate, Zugriff direkt (kein Index)
    * `interface OrderedSet<E>`
      Zugriff benutzt Vergleichsmethode

Maps — Überblick

- `interface Map<K,V>` Abbildung von K nach V

  keine Duplikate (partielle Funktion, endliche Definitionsbereich), Zugriff direkt

  - `interface OrderedMap<K,V>`
    Zugriff benutzt Vergleichsmethode von K

Collections-Dokumentation:

- `http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html`
- Source: `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/oo/j2sdk1.5.0/src/`
- in Eclipse (ab 3.1, mit JDK1.5.0) *open declaration*

Collection/Iterator

```
interface Collection<E> {
    int size (); boolean isEmpty ();
    boolean add (E o);
    boolean addAll (Collection<? extends E>
    Iterator<E> iterator();
}
interface Iterator<E> {
    boolean hasNext ();
    E next ();
    void remove ();
}
```

Vereinfachte For-Schleife

alt:

```
Collection<E> c = ... ;
for ( Iterator <E> it = c.iterator ()
    ; it.hasNext () ; ) {
    E x = it.next ();
    ...
}
```

neu:

```
Collection<E> c = ... ;
for ( E x : c ) {
    ...
}
```

interface List<E>

```
interface List<E> extends Collection<E> {
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
}
```

Implementierungen:

- `ArrayList<E>`, Zugriff über Index schnell, Einfügen langsam (wg. Kopie)
- `LinkedList<E>`, Index-Zugriff langsam, Einfügen schnell (kein Kopieren)

Übung (RTFC): Such- bzw. Kopierbefehle suchen

http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/oo/j2sdk1.5.0/src/

Iteratoren für Listen

Iterator bewegt *Cursor*, dieser steht immer *zwischen* Elementen

```
interface List<E> extends Collection<E> { ..
    ListIterator<E> listIterator ();
}
interface ListIterator<E> {
    boolean hasNext (); E next ();
    boolean hasPrevious (); E previous ();
    int nextIndex (); int previousIndex ();
    void remove (); // lösche das zuletzt ge
    void set (E o); // ersetze das zuletzt g
    void add (E o); // zwischen Cursor und p
}
```

Übung: eine Folge von remove, set, add ausführen.

`interface Set<E>`

enthält keine Duplikate (bzgl. `equals()`)

```
interface Set<E> extends Collection<E> { ..
```

wichtige, sehr effiziente Implementierung:

```
class HashSet<E> implements Set<E> { .. }
```

Hashing

Idee: Objekt o wird abgebildet auf Hash-Wert $h(o)$ und gespeichert in $t[h(o)]$.

Problem: $o \neq p$, aber $h(o) = h(p)$. (Kollision)

Lösungen:

- in der Tabelle (anderen Platz suchen)
- außerhalb der Tabelle (Tabellen-Einträge sind Listen)

Übung (RTFC): welche Variante wurde gewählt?
`http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/oo/j2sdk1.5.0/src/`

`interface OrderedSet<E>`

Operationen wie `Set<E>`, aber benutzt Ordnung auf Elementen.

(Iterator liefert aufsteigend geordnete Folge.)

Wichtige Implementierung: `TreeSet<E>` liefert balancierte Suchbäume.

Übung (RTFC): wie sind die balanciert?

`http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/oo/j2sdk1.5.0/src/`

Ordnungen (I)

Welche Ordnung wird verwendet?

- `OrderedSet<E> s = new TreeSet<E> ();`

  benutzt „natürliche“ Ordnung

```
interface Comparable<E> {
    int compareTo (E o);
}
```

Beachte: Ordnung muß *konsistent mit* `equals()` sein — was bedeutet das genau? Extrahiere aus der Dokumentation eine formale Beschreibung, benutze Eigenschaften von Relationen (siehe 1. Semester)

Übungen zu Collections (KW 17)

(Fortsetzung der Übung zu Suchbäumen)

- Methode `add` (Einfügen in Suchbaum), `contains` (Enthaltensein im Suchbaum).

```
class Tree<E extends Comparable<E>> { ...
        public void add (E x) { ... }
        public void addAll (Collection<E> c
        public boolean contains (E x) { ...
}
class Entry<E> { ...
        static <E extends Comparable<E>> En
}
```

und teste:

```
class TreeTest {
        public static void addTest () {
                System.out.println ("addTes
                String [] words = { "foo",
                Tree<String> t = new Tree<S
                for (String w : words) {
                        System.out.println
                        t.add (w);
                        System.out.println
                }
        }
}
```

- wie lauten die Spezifikationen von `add` und `contains`? Hinweis: reicht der folgende Test aus?

```
class TreeTest { ...
        public static void containsTest ()
                System.out.println ("contai
                List<Double> l = generate (
                Tree<Double> t = new Tree<D
                t.addAll (l);
                for (Double d : l) {
                        boolean result = t.
                        System.out.println
                }
        }

        static List<Double> generate (int s
                List<Double> result = new L
```

```
                for (int i=0; i<size; i++)
                    result.add (Math.rand
                }
                return result;
            }
    }
```

- Wie beweist man, daß die Implementierungen tatsächlich die Spezifikationen erfüllen? Welche Eigenschaften der durch `equals` und `compareTo` definierten Relationen werden bei `add` und `contains` vorausgesetzt? Beantworte anhand des Quelltextes!

- liefere Liste der Schlüssel in Inorder-Reihenfolge, soll so benutzt werden:

```
Tree<Integer> t = Tree.full (2);
List<Integer> l = t.toList();
System.out.println (l);
```

Deklaration (sichtbar):

```
class Tree<E extends Comparable<E>> {
        ...
         public List<E> toList () {
                List <E> l = new LinkedList
                Entry.addToList (l, root);
                return l;
        }
}
```

Implementiere die passende Methode:

```
class Entry<E> { ...
        static <E> void addToList (List<E>
}
```

- Benutze `add` zum Sortieren (Ausgabe mit `toList`).

```
class Sort {
    public static <E extends Comparable<E>>
                    tree_sort (List <E> input
}
```

- vergleiche die Leistung der selbstgebauten Suchbaum-Implementierung:

```
List<Double> in = generate (10);
List<Double> out = Sort.tree_sort(in);
```

```
System.out.println ("out: " + out);
```

mit der Implementierung aus der Bibliothek:

```
Set<Double> s = new TreeSet<Double> ();
s.addAll (in);
System.out.println ("s: " + s);
```

(für längere Eingaben)

Wie sind die offiziellen Suchbäume balanciert?

interface Map<K,V>

Abbildung (partielle Funtion mit endlichem Definitionsbereich) von K nach V

```
interface Map<K,V> {
  int size(); boolean isEmpty();
  V get (K key);
  V put(K key, V value);
  Set<K> keySet();
  Collection<V> values();
}
```

Map (II)

```
interface Map<K,V> { ...
  Set<Map.Entry<K, V>> entrySet();

  interface Entry<K,V> {
    K getKey ();
    V getValue ();
  }
}
```

Implementierung: `HashMap<K,V>`

```
interface OrderedMap<K,V> { .. }
```

Implementierung: `TreeMap<K,V>`

Aufgabe zu Collection/Map

die 10 häufigsten Wörter feststellen

```
Map <String,Integer> counter =
    new HashMap <String,Integer> ();
// TODO: counter füllen, siehe nächste Folie

class Comp<K extends Comparable<K>,
           V extends Comparable<V>>
    implements Comparator<Map.Entry<K,V>>
{ .. } // TODO: Vergleich nach V
```

```
SortedSet<Map.Entry<String,Integer>> t =
    new TreeSet<Map.Entry<String,Integer>>
        (new Comp<String,Integer>());
t.addAll (counter.entrySet ());

int clock = 10;
for (Map.Entry <String,Integer> e : t) {
    System.out.println (e);
    if (--clock < 0) break;
}
```

Datei lesen

...und in Wörter zerlegen:

```
import java.io.*;

Reader r = new FileReader ("foo.bar");
StreamTokenizer st = new StreamTokenizer (r)
while (StreamTokenizer.TT_EOF != st.nextToke
    switch (st.ttype) {
    case StreamTokenizer.TT_WORD: {
        System.out.println (st.sval);
} } }
```

Hausaufgabe: Anagramme

(bis 24. Mai $=$ vor der nächsten Vorlesung)

Finden Sie aus einem Eingabestrom alle Mengen von Anagrammen (Wörter, die durch Buchstabenvertauschungen auseinander hervorgehen, z. B. $\{$nebel, leben$\}$)

Lösungsplan:

- gelesenes Wort $u = \mathrm{nebel}$, sortiere alphabetisch $v = \mathrm{beeln}$, benutze dazu geeignete Bibliothek/Methode (nicht selbst programmieren)

- konstruiere daraus `Map<String,Set<String>> f`, so daß schließlich `f.get("beeln")` $= \{\mathrm{nebel}, \mathrm{leben}\}$

Java-Generics formal (3. 5.)

Generics I (Wdhlg.)

Generische Klassen:

```
class Foo <S, T extends Bar> { .. }
```

- Innerhalb von `{ .. }` sind `S, T` wie Typnamen verwendbar.

- `Foo` ist eine Klassen-*Schablone*, erst durch *Typ-Argumente* wird daraus eine *Klasse*.

Generics II (Wdhlg.)

Generische Methoden:

```
static <T> void print (Collection<T> c)
  { .. }
```

- ist eine Methoden-Schablone

- Compiler rechnet Typ-Argumente selbst aus

Schablonen-Argumente gibt es nur bei Typen, also in Deklarationen, und nicht in Ausdrücken/Anweisungen.

Beachte aber Konstruktoren!

C++ (KW 21)

Geschichte: C

ca. 1970 Kernighan/Ritchie: Sprache C

für die maschinenunabhängigen Teile von Unix.

(bis heute: C $=$ portable assembly language)

Volle Kraft zurück: von Java zu C

- `class` heißt `struct`
- die Methoden stehen nicht in,
  sondern nach der Klasse
- Speicherverwaltung ist nur teilweise automatisch

(jetzt sind wir bei C++, weiter zu C:)

- es gibt keine Vererbung
- alle Methoden sind `static`
- gar keine automatische Speicherverwaltung

Eigenschaften von C

- hat:
  - blockstrukturierte Programme
  - blockstrukturierte Daten (`struct`)
- hat nicht:
  - Module (nur `#include`)
  - Typsystem („alles ist int“) (Char, Boolean, Zeiger)
  - Polymorphie (nur Macros)
  - Datenabstraktion
    (aber: Schnittstellen $\rightarrow$ Funktionszeiger)

Virtual Function Tables in C

Schnittstelle:

/usr/src/kernel-source-2.4.18/include/linux/

```
struct file_operations {
  int (*open) (struct inode *, struct file *
  ..
};
```

Implementierung:

/usr/src/kernel-source-2.4.18/fs/ext2/file.c

```
struct file_operations ext2_file_operations = {
  read:           generic_file_read,
  write:          generic_file_write,
  open:           generic_file_open,
};
```

Geschichte: C++

ab 198?: Bjarne Stroustrup: erweitert C um *Klassen*, dann *virtuelle Funktionen* (= vereinfachte Syntax für Benutzung von Funktionen in *structs*)

Objekt:

```
struct stack {
  int s [SIZE];
  int top;
  struct stack_vtable
     * vtable;
};
```

(virtuelle) Klasse:

```
struct stack_vtable {
  void (* push)
       (struct stack *, int
  int  (* pop )
       (struct stack *);
};
```

Methoden-Implementierung:

```
void stack_push (struct stack * this, int c)
  {  ( this->s ) [ ( this->top )++ ] = c; }
```

Methoden-Aufruf (`st.push (i);`)

```
  st -> vtable -> push (st, i);
```

versucht ansonsten strenge Abwärts-Kompatibilität zu C (Quelltext *und* Objekt-Code!)
Das ist einerseits Grund für weiter Verbreitung der Sprache, andererseits eine schwere Einschränkung.

- C++ hat:
  - verbessertes Typsystem
  - Namespaces, (virtuelle) Klassen zur Modularisierung
  - Templates (für Klassen und Funktionen) zur generischen Programmierung

C++ oder Java

softwaretechnisch ist Java einfach „10 Jahre voraus“:

- Module (Klassen, Packages)
- Interfaces
- genauere Typen für generische Funktionen/Klassen
- vollautomatisches Speichermanagement

auch Java kann zu schnellem nativem Code kompiliert werden.

bleiben „nur“ diese Argumente für Benutzung von C/C++

- maschinennahe Programmierung

  (bei genauer Betrachtung höchst selten nötig,
  für viele Anwendungen gibt es geeignete APIs)

- Legacy-Code (Applikationen, Bibliotheken)

  (der muß weiter gewartet und benutzt werden, weil
  bereits viel darin investiert wurde)

Moderner Zugang zu C++

nach Andrew Koenig/Barbara Moo: Accelerated C++, Adison-Wesley, 2000 (*), erklärt im wesentlichen

- die generischen Klassen (Container, Iteratoren)/Algorithmen aus der Standardbibliothek (STL)

- und Sprach-Details nur bei Bedarf.

„standard“ bedeutet auch: *keine* herstellerspezifischen Bibliotheken (sogenannte „foundation classes“)

(*) deutsch 2003, der Übersetzer sollte allerdings nochmal gründlich die Regeln zur Kommasetzung wiederholen

Literatur

- SGI: Standard Template Library Programmer's Guide (ältere Version), `http://www.sgi.com/tech/stl/index.html`
- Andrew Koenig and Barbara E. Moo: `http://acceleratedcpp.com/`
- Mathias Linke: STL (Vortrag im Seminar Software-Entwicklung), `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/se/linke/`
- GCC (GNU Compiler Collection) `http://gcc.gnu.org/`, GNU Standard C++ Library

`http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-3.3/index.html`

- Marshall Cline: C++ FAQ (Lite), `http://www.parashift.com/c++-faq-lite/`

Beispiel: Wörter sortieren

```
#include <algorithm>
#include <list>
#include <string>
#include <iostream>
int main (int argc, char ** argv) {
  std::list<std::string> l;
  for ( std::string s ; std::cin >> s ; ) {
    l.push_back (s);
  }
  show (l); l.sort (); show (l); return 0;
}
```

beachte: Imports, Objekte, Methoden, generische Container, Namespaces `::`, Operator >>

Beispiel: Wörter sortieren (II)

```
template<class T>
void show (std::list<T> l) {
  for ( std::list<T>::iterator it
          = l.begin ()
      ; it != l.end ()
      ; ++it )
  {
    std::cout << *it << std::endl;
  }
}
```

beachte: generische Funktion, Iterator, Operator <<

Übung (KW 21)

- das vorige Beispiel (Wortliste sortieren) kompilieren und ausführen. Liest von `stdin`.

```
emacs -f server-start &
emacsclient Sort.cc &
gmake Sort # sollte g++ aufrufen (ohne Make
./Sort < Sort.cc
```

- Entwicklungsumgebung („retrocomputing“):
  `bash, emacs, gmake, gcc/g++, gdb`

- Emacs kann alles, außer Tee kochen: probiere
  `M-x hanoi, M-x doctor, M-x gomoku, M-x dunn`

- Wortliste in umgekehrter Reihenfolge ausgeben (benutze `reverse_iterator`)

- Wortliste nach der Wortlänge sortieren

  (schreibe eigene Vergleichsfunktion)

  beachte Art und Zeitpunkt der Typprüfung

Mehr zu C++

(Separate) Kompilation

- kompiliert wird jeweils *eine Quell-Datei* zu *einer Objekt-Datei*

  (`g++ Foo.cc` erzeugt `Foo.o`, `g++ Bar.cc` erzeugt `Bar.o`)

- mehrere Objekt-Dateien (und ggf. Bibliotheken) werden zu einer *ausführbaren Datei* gelinkt

  (`g++ Foo.o Bar.o -o Foo` erzeugt `Foo`).

Header

- alle öffentlichen Deklarationen in *Header-Datei* auslagern (`Foo.h` zu `Foo.cc`)
- jede Quell-Datei importiert Header-Dateien nach Bedarf (und *immer* ihren eigenen Header)
- beachte unterscheidliche Syntax für imports:
  - System-Header: `#include <string>`
  - eigene Header: `#include "Foo.h"`
- schütze Header gegen Mehrfach-Import: in `Foo.h`:

```
#ifndef FOO_H
#define FOO_H
...
#endif
```

Aufgabe zu Imports

für Gnu-Glibc
`http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-3.4/index.html`

- welche Dateien werden tatsächlich importiert bei `<list>` und `<map>`?
- wie wird der zugrundeliegende Suchbaum einer `map` balanciert?

namespace, using

in `<list>` steht:

```
namespace std {
  template<typename _Tp>
  class list { .. }
}
```

Benutzung dann so:

```
#include <list>
std::list<int> foo;
```

oder so:

```
#include <list>
using namespace std;
list<int> foo;
```

Sichtbarkeit

- Reihenfolge von Deklarationen ist wichtig!
- Sichtbarkeit einer Deklaration beginnt mit ihrem ersten Auftreten im Quelltext.
- Vorherige Bezüge sind nicht erlaubt (bzw. führen zu „unverständlichen“ Fehlermeldungen, weil der Kompiler dann annimmt, daß alle Typen `int` sind)

- für gegenseitige Verweise sind Vorwärts-Deklarationen möglich:

```
struct A;
struct B { struct foo * A; };
struct A { struct bar * B; };
```

(das geschieht für Funktionsköpfe in Header-Dateien!)

```
Foo.h:
string f ();

Foo.cc:
string f () { return "foobar"; }
```

Container in der STL

- sequentielle Container:
  string, (wstring), list, vector, (dequeue)
- assoziative Container:
  set, (multiset), map, (multimap) . . .

beachte Unterschiede zu Java/Collections

- in STL gilt auch `map` als Container
- in STL gibt es nur konkrete Typen (Implementierungen), keine Schnittstellen
- Vorbedingungen (z. B. „Elementtyp muß Vergleichsoperation haben“) können nicht programmiersprachlich formuliert werden

Container-Methoden und -Typen

Pflicht:

- `void clear ();` löscht alles
- `size_type size ();` Größe,
  `bool empty ();` ist leer?
- `iterator begin ();` `iterator end ();`

benutzt Typen

- `reference`, `value_type`
- `iterator`, `reverse_iterator`
- `difference_type`, `size_type`

Sequentielle Container-Methoden

- `reference at (size_type n);`
  `reference operator[] (size_type n);`
  indizierter Zugriff (vector)
- `reference front();` `reference back();` (list, vector)
- `push_front (value_type & x);` vorn anfügen (list)

  `push_back (value_type & x);` hinten anfügen (list, vector)
- `pop_front ();` vorn löschen (list)

  `pop_back ();` hinten löschen (list, vector)

Iteratoren – Benutzung

Container hat Methoden:

- `iterator begin(); iterator end();`,

  `reverse_iterator rbegin();`
  `reverse_iterator rend();`,

- `iterator insert (iterator p, value_type & x`

  fügt x vor p ein. Resultat: Iterator, der auf x zeigt.

  (bei vector: spätere Iteratoren werden ungültig.)

Iteratoren – Eigenschaften

Es gibt diese Arten von Iteratoren:

- Input: nur `++`, nur einmal lesen (`*it`)
- Output: nur `++`, nur einmal schreiben (`*it`)
- Forward: nur `it++`,

  beliebig lesen und schreiben über `*it`
- Bidirectional: wie Forward, zusätzlich `it--`
- Random access: wie Bidirectional,

  zusätzlich `operator[]`

  und Index-Rechnungen `+`, `-` mit `difference_type`

Listen

- `void reverse();` spiegeln
- `void sort();` sortieren (bzgl. `operator<`)
- Sortieren nach selbst gewählter Ordnung:

```
template<typename _StrictWeakOrdering>
void sort(_StrictWeakOrdering __comp);
```

wie ist der Typ von `__comp` deklariert?

Liste (II)

```
template<typename _Tp>
template <typename _StrictWeakOrdering>
void list<_Tp>::
merge(list& __x, _StrictWeakOrdering __comp)
{
  iterator __first1 = begin();
  iterator __last1 = end();
  iterator __first2 = __x.begin();
  iterator __last2 = __x.end();
  while (__first1 != __last1 && __first2 != __last2)
    if (__comp(*__first2, *__first1)) { .. }
}
```

Typprüfung erst *nach Instantiierung*

(späte Fehler-Erkennung)

Assoziative Container

definieren zusätzlich Typ `key_type`,
Inhalt ist immer bgzl. dessen Ordnugsrelation geordnet.

- `iterator find (key_type & x);`
- `iterator insert (value_type & x);`

Implementierungen:

- set: `value_type = key_type`
- map:
  `value_type = pair<key_type,mapped_type>`

Varianten: multiset, multimap

map

Abbildung mit endlichem Definitionsbereich, aufgefaßt als Menge von Paaren (aufsteigend geordnet nach der ersten Komponente)

D. h. die Iteratoren einer map liefern:

```
template <S,T>
class pair { S first, T second }
```

Zugriff auf eine map:

```
mapped_type & operator[]
      (const key_type & x);
```

falls `x` nicht vorhanden, wird es eingefügt (und sein Wert wird default-initialisiert)

Aufgabe zu map

zehn häufigste Wörter des Eingabestroms feststellen

```
#include <algorithm>
#include <map>
#include <list>
#include <string>
#include <iostream>

using namespace std;

// paar mit vertauschten komponenten erzeuge
template <class S, class T>
... flip (pair <S,T> p) {
  return ....
```

```
}

template<class S, class T>
ostream & operator<< (ostream & os, const pa
  return os << "(" ... ;
}

int main (int argc, char ** argv) {

  map<string,int> m;
  for (string s ; cin >> s ; ) {
    m [s] ++;
  }
```

```
  // elemente von m geflippt in liste eintra
  list<pair<int,string> > l;
  for ( ; ; ) {
    ...
  }

  // liste sortieren
  // liste spiegeln

  l.resize (10); // liste kuerzen

  // liste ausgeben

}
```

Referenzen, Const

Überladen von Operatoren

```
template<class S, class T>
ostream & operator<<
  (ostream & os, const pair<S,T> p)
{
  return os << "(" << p.first
            << "," << p.second << ")";
}
```

Beachte Benutzung von Referenzen.

Zeiger

Zu jedem Typ `T` gibt es den Typ *Zeiger auf* `T`.

Deklaration einer Variablen mit Zeigertyp durch `*` vor dem Namen (*nicht* vor dem Typ).

- Deklaration:
```
struc T { int foo; string bar; };
T o; // ein Objekt
T * p = & o; // p zeigt auf o
```
- das Objekt, auf das p zeigt: `*p`
- ein Attribut (member) dieses Objekts: `*p.foo`
- abgekürze Notation dafür: `p->foo`

Iteratoren verhalten sich wie Zeiger
(d. h. `operator*` ist überladen)

Referenzen

`T & n` bedeutet: der Name n enthält einen Verweis (reference) auf ein Objekt vom Typ `T`.
wird bei Unterprogrammen benutzt, um:

- Kopier-Aufwand zu sparen

- Information zurückzugeben

```
template<class S, class T>
ostream & operator<<
    (ostream & os, const pair<S,T> p)
{ return os << "(" << p.first
            << "," << p.second << ")";  }
```

Vergleich Zeiger/Referenzen

- beides sind Verweise auf Objekte
- eine Referenz kann genau einmal de-referenziert werden (es gibt keine Referenz auf eine Referenz), die De-referenzierung ist implizit.
  ein Zeiger kann oft de-referenziert werden (es gibt Zeiger auf Zeiger), De-referenzierung durch Stern-Operator
- mit Referenzen kann man nicht rechnen
  es gibt Zeiger-Arithmetik (vgl. Iteratoren mit ++)

Moral:

- Für Verweise *immer* Referenzen benutzen
  (und statt Zeigern: Arrays oder Container)

const

Wenn ein Name durch `const` deklariert wird, darf das bezeichnete Objekt nur gelesen, aber nicht geschrieben werden.

Beispiel: der `value_type` einer `map` ist tatsächlich `pair<const key_type, mapped_type>`

Wdhlg Iteratoren

find (Input)

```
template <class In, class X>
In find (In begin, In end, const X& x) {
  while (begin != end && *begin != x) {
    ++begin;
  }
  return begin
}
```

copy (Output)

```
template <class In, class Out>
In copy (In begin, In end, Out target) {
  while (begin != end) {
    *target++ = *begin++;
  }
  return target
}
```

Streams und Iteratoren

Zu jedem Ein/Ausgabe-Strom gehört ein Ein/Ausgabe-Iterator:

```
vector<int> v;

copy (istream_iterator<int>(cin),
  istream_iterator<int>(), // bedeutet EOF
  back_inserter (v));

copy (v.begin(), v.end(),
  ostream_iterator<int>(cout
      , " "); // getrennt durch Leerzeichen
```

replace (forward)

```
template ...
... replace ( ... ) {
  while ( begin != end ) {
    if (*begin == x) {
      *begin = y;
    }
  }
}
```

reverse (Bidirektional)

```
template ...
void swap (...) {

}

template ...
... reverse ( ... ) {
  while ( begin != end ) {
    ... swap (... ) ...
  }
}
```

binary search (Random-Access)

```
template <class It, class X>
bool binary_search (...) {
  if (begin < end) {
    It mid = begin + (end - begin)/2;
    if ( x < *mid ) {
      return binary_seach (begin, mid);
    } else if ( x > *mid ) {
      return binary_seach (mid, end);
    } else { return true; }
  } else { return false; }
}
```

- finde äquivalente Version ohne Rekursion (stattdessen mit `while`)

- warum `begin + (end-begin)/2` und nicht

```
(begin + end)/2 ?
```

Übung zu Iteratoren

Vervollständigen Sie die angegebenen Code-Stücken und testen Sie diese.

Hinweis: da es die Namen (`replace, reverse` usw.) bereits im default-namespace gibt, benutzen Sie einen eigenen namespace:

```
#include <list>

namespace mein {
  template ...
  void swap (...) { ... }

  template ...
  ... reverse ( ... ) { ... }
```

```
  }
}

int main () {
  mein::reverse (...); ...
}
```

Definition neuer Typen

Typen: `struct`

- *Attribute* (members)
- und *Methoden* (member functions)

```
#include <vector>
#include <string>
struct Student {
  std::string name;
  std::vector<double> zensuren;
  double note () const;
}
```

const-Methode darf „ihr“ Objekt nicht ändern.

Nicht-const-Methode nur für nicht-const-Objekte

Sichtbarkeit von Deklarationen

- public: sieht jeder
- private: nur innerhalb der class/struct sichtbar

typische Benutzung:

```
class C {
    int x;
public:
    get_x ();
}
```

Defaults:

- struct: members sind public
- class: members sind private

Ort von Deklarationen

- struct/class mit Deklarationen von Attributen und Methoden immer im header (.h)
- Implementierung der Methoden normalerweise in .cc
- in headers ggf. `#include`, aber *kein* `using`

Beispiel (Student.cc):

```
#include "Student.h"

double Student::note () { ... }
```

Konstruktoren

sind besondere Methoden, Name $=$ Typname, kein Resultat.

```
class Student { ...
public:
    Student ();
    Student (std::istream&);
}
```

können *nicht* explizit aufgerufen werden. Benutzung in Deklarationen:

```
Student s;
Student s (cin);
```

Initialisierungen

in Konstruktor nach Doppelpunkt

```
struct Student {
  int matrikel;
  string name;

  Student (int m, string s)
    : matrikel (m), name (s) { }
}
```

ist die einzige Möglichkeit, const-member zu setzen.

Speicher-Management

Platz für Daten

- automatisch: `{ int x; ... }`
  Objekt lebt nur innerhalb seines Blocks
- statisch: `{ static int x; ... }`
  Objekt lebt „ewig “(aber nur einmal)
- dynamisch:
  `{ int * p = new int (42); ... ; delete p}`
  Programmierer bestimmt Lebensdauer (durch `new`/`delete`) selbst $\rightarrow$ flexibel, aber gefährlich

für Felder: `T * a = new T [n]; ... delete [] p;`

Arten von (Kon)Struktoren

- Default-Konstruktor / Konstruktor mit Argumenten:

```
class T {
  T () { .. }
  T (int s) { .. }
}
T v; // Benutzung
T v(100);
```

- Copy-Konstruktor: (bei Übergabe an und von Funktion, auch in Initialisierungen)

```
class T { T (const T & x) { .. } }
T a;
f (a); // benutzt Copy-Konstruktor
```

- Zuweisungs-Operator:

```
class T {
  T & operator= (const T & x) { .. }
}
T a; T b;
a = b; // Benutzung
```

Beachte:

– Zuweisung `a = b;` benutzt `operator=`
– Initialisierung `T a = b;` benutzt Copy-Konstruktor.
– Deklaration `T a;` benutzt (Default-)Konstruktor

- Destruktor:

```
class T { ~ T () { .. } }
```

aufgerufen, falls Lebenszeit eines Objekts endet

– implizit (am Block-Ende) bei automatischen,
– durch `delete` für dynamische

(Default-)Implementierungen

Dreierregel: Destruktor, Copy-Konstruktor, Zuweisungs-Operator immer *komplett* implementieren oder *gar nicht*.

Der Compiler erzeugt sinnvolle Defaults, die für die Fälle ausreichen, bei denen keine permanenten Ressourcen alloziert werden.

Aufgaben zu Konstruktoren

- definiere `struct Student` mit Matrikelnummer (int), Vorname, Name (const string),

- implementiere und teste
  - Konstruktor
    soll Matrikelnummer automatisch erzeugen (nächste freie Nummer: benutze statisches Attribut)
    beachte die `const` für Vorname, Name
  - Copy-Konstruktor
    soll Matrikelnummer ändern
  - Zuweisungs-Operator,
    soll Matrikelnummer überschreiben
  - Destruktor

Jeder Kon/De-struktor soll eine Nachricht schreiben, z. B.

```
cerr << "Copy-Konstruktor" << endl;
```

Aufgaben (abzugeben sind jeweils Programm-Fragment *und Auswertung/Beantwortung*)

- wie werden die const-Attribute initialisiert?
- deklariere je einen automatischen, statischen und dynamischen Studenten. In welcher Reihenfolge werden Konstruktoren/Destruktoren gerufen?
- deklariere zwei automatische Studenten (in einem Block). in welcher Reihenfolge werden Destruktoren aufgerufen?
- welcher Konstruktor wird bei Argument-Übergabe an

Funktion benutzt?

Zusammenfassung

Objektorientierung

konkreter Datentyp

- Gruppierung von Daten (Attribute) und Operationen (Methoden)
- Verstecken von Implementierungen, Veröffentlichen von Schnittstellen

Implementierungsvererbung

abstrakter Datentyp

- Formalisieren von (gemeinsamen) Schnittstellen
- Schnittstellenvererbung

OO historisch

- Simula 67 (Kristen Nygaard, Ole-Johan Dahl, ab 1962)
- Smalltalk (Alan Kay, 1980, Xerox PARC)
- C++ (Bjarne Stroustrup, 1985, AT& T)
- Java (James Gosling, 1994, Sun)
- Eiffel (Bertrand Meyer), Self, . . .

Wiederverwenden und parametrisieren

man möchte (Software-)Produkte grundsätzlich

- einmal entwickeln
- mehrfach verwenden

Wiederverwendung dabei nicht unter identischen Bedingungen

$\longrightarrow$ Software muß *parametrisiertes Schema* sein.

Bei Verwendung werden im Schema die Parameter durch konkrete Werte ersetzt.

SoftwareSchemata und -Parameter

- klassisch (seit Algol, Pascal, . . .)

  Schema ist Unterprogramm, Parameter sind Daten

- wünschenswerte Erweiterungen:
  - als Parameter-Daten auch Unterprogramme
    $\longrightarrow$ funktionales Programmieren
  - als Parameter auch/nur Typen
    $\longrightarrow$ polymorphe Unterprogramme
  - als Schema auch Typ-Deklarationen
    $\longrightarrow$ polymorphe Datentypen (Klassen)

  . . . das sind gar keine Erweiterungen, sondern
  Aufhebungen von (willkürlichen) Beschränkungen

Funktionales C++: generate

```
int gen () { static int c = 0;  return ++c;

list<int> l (n);
generate (l.begin(), l.end(), gen);
```

Implementierung (in <algorithm>)

```
template<typename _ForwardIter, typename _Generator>
  void generate(_ForwardIter __first,
                _ForwardIter __last, _Generator __gen)
    {
      for ( ; __first != __last; ++__first)
        *__first = __gen();
    }
```

Funktionales C++: accumulate

```
int add (int x, int y) {  return x + y; }


return accumulate (l.begin(), l.end(), 0, ad
```

Implementierung (in <numeric>)

```
template<typename _InputIterator, typename _Tp,
         typename _BinaryOperation>
  _Tp  accumulate(_InputIterator __first,
         _InputIterator __last,
         _Tp __init, _BinaryOperation __binary_op)
    {
      for ( ; __first != __last; ++__first)
        __init = __binary_op(__init, *__first);
      return __init;
    }
```

Funktionales Haskell

```
main = print $ foldr (+) 0 $ take 100
     $ unfoldr ( \ c -> Just (c, c+1) ) 1
```

Implementierung (in `Prelude, List`):

```
unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
unfoldr f b = case f b of
    Nothing    -> []
    Just (a,b) -> a : unfoldr f b
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x : xs) = f x (foldr f z xs)
```

Funktionen sind *Bürger erster Klasse*

(dürfen lokal definiert werden, Argument und Resultat anderer Funktionen sein)

Funktionales Java: Wrapper

Funktionen (Unterprogramme) werden benötigt bei

- generischen Algorithmen für Container
- . . . zum Sortieren (Vergleichs-Funktion)
- zur Ereignis-Behandlung

In Java gibt es überhaupt keine „Funktion/Unterprogramm an sich“,
sondern nur *Methoden von Klassen*.

- Funktion wird Methode eines *Wrapper-Interfaces*
- statt Funktion benutze Wrapper-Objekt
- Beispiele: `Comparator, ActionListener`

Polymorphie

(= Typen als Parameter) Implementierungen:

- Java (< 1.5): Programmierer muß Typ-Information entfernen (Promotion/Cast zu/von Object): schreibt ungetyptes Programm
- Java 1.5 (generics): Compiler prüft Typen (dann: löscht Typen) und erzeugt Code *für Schema*
- C++ (templates): Compiler *instantiiert* (ersetzt Parameter durch Wert), prüft Typen und erzeugt Code für *Verwendung des Schemas*

C++ erlaubt auch *Daten* als Schablonen-Parameter:

```
template <typename E, int size> class Store
```

Polymorphie (II)

grundsätzliche Fragen an Programmiersprache:

- welche Möglichkeiten für Abstraktionen gibt es?
- kann man mit diesen vernünftig umgehen?

Beispiele:

- C: Anweisungen $\rightarrow$ Unterprogramme,
  (Zeiger auf) Unterprogramm als Datum benutzen
- C++: Code $\rightarrow$ Template für Code,
  keine Templates für Templates,
- funktionale Programmierung (Haskell):
  Funktionen höherer Ordnung *sind* Funktionen
  Typkonstruktoren (`List ..,Tree ..`) können
  Schnittstellen implementieren

Orthogonalität

Polymorphie über *alle* Typen?

Es gibt Unterschiede zwischen atomaren Typen, Arrays, Objekten, Funktionen

Beispiel: Sortieren.

- C++: `bool operator< (T, T)`
- Java: `T implements Comparable<T>`

  nicht für atomare Typen möglich

  deswegen Wrapper-Klassen (`Integer`, `...`),
  mit Auto(un)boxing

Modulsysteme

- C++: jein (Header/Implementierung),

  implementiert durch Präprozessor,

  spezielle Behandlung für System-Header

- Java: Klassen/Pakete

  keine Trennung von Schnittstelle und Implementierung (Ausweg: Interfaces)

Beide: Linker-Unterstützung für vorkompilierte Programmteile

Sprachdefinitionen

- C++: ist ISO-Standard: hersteller-unabhängige Übereinkunft

  Zugriff auf Standard ist nicht kostenlos

- Java: sieht „offen“ aus, ist aber im Kern firmenspezifisch

  Implementierungen müssen von Sun „genehmigt“ werden

Typsysteme (OO)

in „klassischen“ OO-Sprachen (C++) deklariert eine Typdeklaration (`class`) *gleichzeitig*:

- einen konkreten Typ T (Menge der Attribute)

  . . . auch als Basis für Ableitungen (mehr Attribute)

- eine Schnittstelle S (Menge der Methoden)

  . . . auch als Basis für Ableitungen (mehr Methoden)

- die Behauptung, daß T die Schnittstelle S implementiert

Bei unvorsichtigem Gebrauch entsteht großes Durcheinander $\rightarrow$ *einzelne Aspekte trennen!*

Typsysteme (Ausblick)

ein besseres Modell ist

- es gibt konkrete Typen $T = \{T_1, T_2, \ldots\}$
- es gibt Interfaces $S = \{S_1, S_2, \ldots\}$
- es bestehen Relationen:
  Implementierungen $\subseteq T \times S$,
  Erweiterungen $\subseteq S \times S$.

Erweiterungen:

- Typen höherer Ordnung ($=$ Funktionen $T^* \to T$), `list< >`
- Interfaces auch für höhere Typen
- mehrstellige Implementierungs-Relationen ($\Rightarrow$ *multiple dispatch*) `(T1, T2) implements S1`

Fazit

Die Wahl einer Programmiersprache für ein Projekt hängt von vielen Faktoren ab, der Programmierer kann diese nur selten beeinflussen.

$\longrightarrow$ Programmierer muß beherrschen:

- gemeinsame Prinzipien von Programmiersprachen
- ihre Realisierung in konkreten Sprachen

Die Entwurfs-Arbeit findet *vor* der Programmierung statt:

> Do not program *in* a language, but *into* a language.