(Objektorientierte) Programmierkonzepte Vorlesung

Johannes Waldmann, HTWK Leipzig

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 Schnittsteller

```
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`

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>()
```

rische Polymorphie und Schnitts

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.js`
- 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 Be-
legungen sind Standard, siehe GNU readline library
`http://cnswww.cns.cwru.edu/~chet/readline/rlt`
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:$PATH
export MANPATH=/home/waldmann/built/man:$MANP
export LD_LIBRARY_PATH=/home/waldmann/built/l
```

(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 ]
```

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/l`

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
```

bezielle Operator-Syntax (Sectior

```
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`?

rtielle Anwendung von Funktion

- 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*:

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

```
module Tree where

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

arianten von Fallunterscheidunge

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
```

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
            }
```

ax: Lokale Deklarationen: let . . . i

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

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

vergleichbar mit (Java):

```
{ T d = W; // Deklaration mit Initialisierung
; 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
```

Übung 12. Woche

acs . . . kann alles außer Tee koch

```
$ 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

Emacs/Hugs als Haskell-IDE

- Emacs für Quelltext im Hintergrund starten:
  `emacs Seminar.hs &`
- Interpreter im Vordergrund und Quelltext laden
  `hugs +Tt Seminar.hs`
  oder `ghci Seminar.hs`, dann `:set +t`
- - Editieren,
  - Emacs:Speichern (C-x C-s),
  - Hugs/GHCi: Re-Laden (:r)

Programmieraufgabe (collatz)

Typ:

```
collatz :: Integer -> [ Integer ]
```

Beispiel:

```
collatz 7 ==>
[7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1
```

Ansatz:

```
collatz 1 = [ 1 ]
collatz x =
    x : collatz ( if undefined then undefined
```

Hinweis: verwende Funktionen `even` und `div`. (das sind Funktionen und keine Operatoren.)

Programmier-Aufgabe (partition)

Typ:

```
partition :: (a -> Bool) -> [a] -> ([a],[a])
```

Beispiel:

```
partition odd [3,1,4,1,5,9,2,6,5,3,5,8,9,7,9]
==> ([3,1,1,5,9,5,3,5,9,7,9],[4,2,6,8])
```

Ansatz:

```
partition p [] = ( undefined , undefined )
partition p (x : xs) =
    let ( yeah, noh ) = partition p xs
    in  if undefined
        then ( undefined  ,  undefined )
        else undefined
```

Suchbäume

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

```
data Tree a = Leaf
            | Node { key :: a
                   , left :: Tree a
                   , right :: Tree a
                   }
    deriving Show
        -- entspr. automatischer Deklarati
        -- der Java-Methode toString
t :: Tree Int
t = Node { key = 5
```

Programmtransformationen (I)

```
sort xs = inorder ( suchbaum xs )

suchbaum [] = Leaf
inorder Leaf = []

sort [] = inorder (suchbaum []) = []
```

Programmtransformationen (II)

```
suchbaum (x : xs) =
   let ( low, high ) = partition ( < x ) xs
   in  Node { key   = x
            , left  = suchbaum low
            , right = suchbaum high
            }
inorder ( Node { } ) =
    inorder (left t) ++ [ key t ] ++ inorder (right t)

sort (x : xs) = inorder (suchbaum (x : xs)) =
   let ( low, high ) = partition ( < x ) xs
   in  inorder (suchbaum low)
         ++ [ x ] ++ inorder (suchbaum high)
sort (x : xs) =
```

Virtuelle Datenstrukturen

```
sort :: [ Int ] -> [ Int ]
sort [] = []
sort (x : xs) =
   let ( low, high ) = partition ( < x ) xs
   in  sort low ++ [ x ] ++ sort high
```

- durch Transformation sind Node/Leaf/Tree verschwunden
  (und das einfache Quicksort bleibt übrig)!
- es ist Aufgabe eines intelligenten Compilers, solche Möglichkeiten zu finden und auszunutzen.

Aufgabe: Rekonstruktion

Schreiben Sie Funktionen

```
preorder :: Tree a -> [a]
inorder  :: Tree a -> [a]
```

die die Schlüssel eines Baumes in der entsprechenden Folge ausgeben sowie eine Funktion

```
recon_pre_in :: [a] -> [a] -> Tree a
```

mit der Eigenschaft:

```
forall t :
    t == recon_pre_in ( preorder t ) ( inorde
```

Beispiel:

```
recon_pre_in
```

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

ster als Funktionen höherer Ordn

```
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 1  ( \ fl k fr -> fl          + fr )
inorder' =
```

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

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
  - `partition`, `append (++)`, `reverse`

laps, Folds und Transformatione

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = f x : map f xs

map (> 2) [1,2,3,4]
    = [False, False, True, True]
```

Dafür gelten Rechenregeln:

```
map f . map g = map (f . g)
```

(für Mathematiker: map ist ein (Endo-)Funktor in der Kategorie der Datentypen)
die sind nützlich bei Programmtransformationen (Vermeiden der Erzeugung temporärer Daten)

. . . das gleiche für Bäume

```
tmap :: (a -> b) -> Tree a -> Tree b
tmap f Leaf = Leaf
tmap f ( n @ Node {} ) =
    Node { key = f ( key n )
         , left = tmap f ( left n )
         , right = tmap f ( right n )
         }
```

wieder gilt: `tmap f . tmap g = tmap (f . g)`

- definiere `tmap` durch `tfold`
- Ergänze die Regel

  `tfold l n . map f = tfold ?? ??`

Fold für natürliche Zahlen

```
data Natural = Zero | Successor Natural

nfold :: .. -> .. -> ..
nfold z s Zero = ..
nfold z s ( Successor x ) = ..

plus x  = nfold x ( \ y p -> Successor p )

times   =
    nfold ( \ y -> Zero ) ( \ x p y -> ... )
power b =
    nfold ( ... ) ( \ x p -> ... )
```

Übung 13. Woche

Übung 13. Woche

- Rekursionsschema für Listen
  - Funktion `lfold` aus Vorlesung kopieren, damit `sum`, `product`, `append`, `reverse` definieren
- Rekursionsschema für Bäume
  - Deklaration `data Tree a = ...`, `tfold :: ...` aus voriger Vorlesung/Übung kopieren.
  - Funktionen `leaves`, `inorder`, `preorder` durch `tfold` definieren und ausprobieren
  - definiere Funktionen `nodes` (Anzahl aller

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 { ... }`

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

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`

omatisches Ableiten von Instanze

```
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

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 Prog
          , ort   = "Reading, Mass."
```

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, emptySet ) ]
        , ist_dran = Schwarz
        }
data Satz = Put Point | Pass | Resign
instanz Zug Havannah Satz where
```

Havannah - Online

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

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

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`

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)

va-Wiederholung: Ausdrucksform

- 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)`

Java-1.5 benutzen

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

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

- 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

ompatibilität zwischen 1.5 und 1

- 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 Methode

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

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

Auto-Boxing

Java unterscheidet:

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

elementare Typen gehören zu keiner Klasse $\longrightarrow$ 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, $\longrightarrow$ boxing und

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 $\longrightarrow$ Preferences $\longrightarrow$ Java $\longrightarrow$ Compiler $\longrightarrow$ compiler compliance level $\longrightarrow$ 5.0

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

Beispiele aus Vorlesung ausprobieren:

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

implementiere polymorphe Funktion

Übung KW 16

Thema: binäre (Such)bäume, vergleiche

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

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`)

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;
```

Java-Collections

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/`
- Source: `http://www.imn.htwk-leipzig.de/~waldmann/e`
- 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> c
    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)

Ü

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 gel
    void set (E o); // ersetze das zuletzt ge
    void add (E o); // zwischen Cursor und pr
```

`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/`

`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/`

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>
        public boolean contains (E x) { ..
}
class Entry<E> { ...
        static <E extends Comparable<E>> E
}
```

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 Map

Sie kennen die Collatz-Folge (vgl. Implementierung in Haskell).
Sie sollen Zahlen mit langer Collatzfolge bestimmen und die Berechnung durch einen Cache beschleunigen.

```
import java.util.*;
class Collatz {
    private static Map<Integer,Integer> lengt
    int static collatz (int start) {
        // wenn start schon im Cache,
        // dann bekannten Wert ausgeben,
        // sonst einen Schritt berechnen
```

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>());
```

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.nextToker
    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

Bewertung der Testate

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

Java-Generics (Das Typsystem)

Generics (Klassen)

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 (Methoden)

Generische Methoden:

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

- ist eine Methoden-Schablone
- Compiler rechnet Typ-Argumente selbst aus
- solche Methode immer `static`
  (siehe nächste Folie)

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

Generics (Methoden II)

```
class Foo {
   static <T> void p (Collection<T> c)
}
class Bar<T> {
   private T foo; void q () { .. this.foo ..
}
```

- generische statische Methode:
  - hat eigene Typparameter
  - die Typparameter der Klasse sind nicht benutzbar
- non-statische Methode:

Java-Grammatik, Deklarationen (I

das ist eine kontextfreie Grammatik, siehe 1. oder 7. Semester (Grundlagen/Compilerbau)

```
Class-Decl ==>
"class" Class-Name "{"
    Variable-Decl ^*
    Method-Decl ^*
"}"
Variable-Decl ==>
    Type-Name Variable-Name
        [ "=" Expression ] ";"
```

Java-Grammatik, Deklarationen (I

```
Method-Decl ==>
    Access-Spec^* Type-Name
        Method-Name "(" Parameters ")" Block
Access-Spec ==>
    public | static | final | ..
Parameters ==>
    ( Variable-Decl "," )^* Variable-Decl
Block ==>
    "{" ( Variable-Decl | Statement )^* "}"
```

Java-Grammatik, Ausdrücke

```
Expression ==>
    Literal
  | Variable-Ref
  | Method-Ref "(" Arguments ")"
  | "new" Type-Name "(" Arguments ")"
  | Expression Operator Expression
  | "(" Expression ")"
Arguments ==>
    ( Expression "," )^* Expression
Method-Ref, Variable-Ref ==>
    ( Class-Name "." )^* Name
  | Expression.Name
```

Java-Grammatik, Anweisungen

```
Statement ==>
    Variable-Ref "=" Expression ";"
  | Expression ";"
  | return [ Expression ] ";"
  | ..
```

Übung: Was fehlt? (ist für Generics unwichtig)

Grammatik-Erweiterungen

generische Klassen definieren:

```
Class-Decl ==> ..
  | "class" Generic-Class-Name
        "<" Type-Parameters ">" "{" .. "}"
Type-Parameters ==>
    ( Type-Parameter "," )^* Type-Parameter
Type-Parameter ==>
  ( Type-Variable | "?" ) [ "extends" Type-Na
```

generische Klassen benutzen:

```
Type-Name ==> Class-Name
  | Generic-Class-Name "<" Type-Arguments ">"
  | Type-Variable
```

Subtyping

(subtyping = Vererbung)
Durch

```
( class | interface ) Foo
    ( implements | extends ) Bar
{ .. }
```

wird `Foo` zu Subtype von `Bar`.
Prinzip der Objektorientierten Programmierung:
*Ü*berall, wo ein Objekt der Basisklasse (hier: `Bar`) erwartet wird, darf auch ein Objekt einer abgeleiteten Klasse (subtype, hier: `Foo`) benutzt werden.

Generics und Subtyping

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls; // ??

lo.add (new Object ());
String s = ls.get (0); // !!
```

D. h., die Annahme

```
S extends T  =>  G<S> extends G<T>
```

führt zu Typfehlern.

Es gilt *nicht*

```
List<String> extends List<Object>.
```

Eingeschränkte Typ-Argumente

```
interface Shape {
  void draw (Canvas c);
}
class Circle implements Shape { .. }
class Rectangle implements Shape { .. }
```

Das folgende

```
void drawAll (List<Shape> xs) {
   for (Shape x : xs) { x.draw (this); }
}
```

ist nicht auf `List<Circle>` anwendbar!

Besser:

```
void drawAll (List<S extends Shape> xs) {
```

(bounded) Wildcards

Statt nicht benutzter Typvariable schreibe `?` (lies: *unknown*)

```
void drawAll (List<? extends Shape> xs) {
   for (Shape x : xs) { x.draw (this); }
}
```

Beachte: `?` ist wirklich *unknown*:

```
void addR (List<? extends Shape> xs) {
    xs.add (new Rectangle ()); // ??
}
```

Mehr zu Wildcards

empfohlene Schreibweise:

```
class Collections { ..
  public static <T>
    void copy
      (List<T> dest, List<? extends T> src);
```

ist besser als:

```
class Collections { ..
  public static <T, S extends T>
    void copy
      (List<T> dest, List<S extends T> src);
```

Generics: Implementierung

- Typ-Argumente müssen zu Typ-Parametern passen: die Schranken der Typ-Parameter erfüllen.
- Typprüfung dann so, also ob jeder instantiierte generische Typ ein eigener konkreter Typ wäre.
- danach werden Typ-Argumente gelöscht, aus instantiiertem generischen Typ wird *roher* Typ (= prähistorische Collection-Klassen).

Vorteile:

- mehr Sicherheit ohne mehr Laufzeitkosten
- generischer Code läuft auf unveränderter

Aufgabe zu Generics

Wie lautet die Deklaration von `q`?

```
public class Typing {
    interface F<A,B> {
            G<A> m ();
            ... q ( ... );
    }
    interface G<A> { H<A> s(); }
    interface H<A> { A r(); }

    static String check (F<H<Integer>,String>
            return y.q(y.m().s()).r();
    }
}
```

Reflection

Beispiele

```
import java.lang.reflect.*;

Class c = baum.Baum.class;

for (Field f : c.getFields()) {
System.out.println (f);
}

for (Method m : c.getMethods()) {
System.out.println (m);
}
```

Polymorphie

eine weitere Form der Polymorphie (Rechnen mit unbekannten Typen)

- allgemein Programmierung „höherer Ordnung“
- dynamisches Ändern von Programmen (z. B. Nachladen von unbekannten Klassen)

Eigenschaften

Reflection. . .

- . . . bezeichnet Ermitteln und Ausnutzen von Typinformationen zur Laufzeit (z. B. Klassen-Signaturen)
- in Java kann diese Information aus class-files bzw. class-Objekten gewonnen werden (Quelltext ist nicht notwendig)
- steht in Widerspruch zu statischer (Compile-Zeit) Typprüfung?
  deren Ziel ist ja, daß man gar keine Laufzeit-Typinformation braucht.

Annotationen

klassische Reflection sieht nur den Quelltext, manchmal möchte man weiter Informationen benutzen:
Quelltext wird annotiert.
Einige Annotationen sind vordefiniert

```
@Deprecated
class Foo { ... }
```

Selbst definierte Annotationen

```
import java.language.annotation.*;

@interface Generate_Doc { }


@Generate_Doc
class Foo { .. }
```

Annotationen mit Argumenten

```
@interface Revision {
  int major () default 1;
  int minor () default 0;
}

@Revision(major=2)
class Foo { .. }
```

zugelassene Werte sind:
int, String, enum, eine andere Annotation oder ein Array davon.

Retention

Annotationen haben eine Verfügbarkeit (retention policy)

- SOURCE (für Programmierer, Compiler, Werkzeuge)
- CLASS (z. B. für Class-Loader) (ist default)
- RUNTIME (für runtime reflection)

```
@Retention(value=RetentionPolicy.RUNTIME)
@interface Generate_Doc { }
```

Annotations zur Laufzeit

```
Class c = o.getClass ();
if (c.isAnnotationPresent(Generate_Doc.class)
   ..
}
```

C#

.NET

Das .NET-Framework von Microsoft:

- .NET-Anwendungen
- Klassenbibliothek
- Laufzeitumgebung (CLR, common language runtime)
- Betriebssystem

siehe
`http://www.microsoft.com/net/developers.mspx`

Mono

- .NET-Portierung als Mono für Linux, siehe `http://www.mono-project.com/`
- gesponsort von Novell
- Installer für Linux `http://www.mono-project.com/Downloads/`

Standards

- ECMA-334 (C#)
  `http://www.ecma-international.org/publicat`
- ECMA-335 (CLI—common language infrastructure)
  `http://www.ecma-international.org/publicat`

ECMA International (European Computer Manufacturers' Association)
C# 2.0 ist inzwischen ISO-Standard, 3.0 ist in Beratung

Sprachen in .NET

- verschiedene Hochsprachen (C#, „Sichtbar Grundlegend“, F#, usw.)
  . . . jeweiliger Compiler $\longrightarrow$
- $\longrightarrow$ gemeinsame Zwischensprache (CIL—common intermediate language: gemeinsame Sprache, Bibliotheken, Typsystem–CTS)
  . . . Lader, Verifizierer, Just-in-time-Compiler $\longrightarrow$
- $\longrightarrow$ Maschinencode

Zwischencode mit Metadaten wird in Assemblies zusammengefaßt, versioniert und signiert.

C#

- entwickelt von Anders Hejlsberg (Borland/Delphi)
- Motivation (geraten, aber offensichtlich)
  - Fortsetzung/Vereinfachung von C++ . . .
  - . . . innerhalb des .NET-Frameworks (Grundlage für modernes MS-Windows)
  - als „Antwort“ auf Java (Sun)

C#-Beispiel

```
using System;
class Hello {
  public static void Main () {
    Console.WriteLine ("Hello World");
  }
}
```

- Datei `Hello.cs` compilieren mit `mcs Hello.cs`, entsteht `Hello.exe`, ausführen mit
  `mono Hello.exe`
- CIL-Schnittstelle: `monop -r:Hello.exe Hello`
- CIL-code: `monodis Hello.exe`

C# und Java

- viele Gemeinsamkeiten — Beispiele: Objekte, Klassen, Methoden; Einfachvererbung für Klassen, Mehrfachvererbung für Interfaces; Exceptions; Annotationen, Reflection; (nachträglich eingebaute) Generics; Boxing; Zwischencode, JIT-Compilation;
- aber auch Unterschiede und Erweiterungen . . .

C# und Java

Unterschiede und Erweiterungen:

- von harmlos . . .
  C# schreibt sowohl Klassen- als auch Attribut- und Methoden-Namen groß
- . . . bis wichtig—Beispiele:
  - Unterschied zwischen Struct (Wert) und Objekt (Verweis),
  - *delegates* $\approx$ Funktions-Objekte,
  - *properties* $\approx$ smart fields

Übung C#/Mono

Im Linux-Pool benutzen: Pfade setzen (in `.bashrc`)

```
export PATH="/home/waldmann/mono-1.1.15/bin:$
export PKG_CONFIG_PATH="/home/waldmann/mono-1
export MANPATH="/home/waldmann/mono-1.1.15/sh
export LD_LIBRARY_PATH="/home/waldmann/mono-1
```

- Hello World,
- ein (fest vorgegebenes) Array von Zahlen sortieren (mit welcher Bibliotheks-Methode?)
- Klasse, mit Attributen, Methoden, Konstruktor

Java-Testat am 8./9. Juni.

Properties

vereinfachte Notation mit dem Ziel:

- intern ist es eine get/set-Methode
- extern sieht es aus wie Attribut-Zugriff

Beispiel

```
class C {
    private int foo;
    int Foo {
        get { return foo; }
        set { foo = value; }
    }
}
```

Indexer

Vereinfachte Notation mit dem Ziel:

- extern sieht es aus wie Array-Zugriff
- intern ist es eine get/set-Methode mit Index
- index-Typ ist nicht unbedingt int

Beispiel (Benutzung)

```
IDictionary<string,int> phone
    = new Dictionary<string,int>();
phone["Jones"] = 234;
phone["Miller"] = 345;
System.Console.WriteLn (phone["Jones"]);
```

(kompilieren mit gmcs anstatt mcs)

Indexer (Implementierung)

Beispiel

```
interface IDictionary<K,V> {
    public V this [K key] {
        get { return ... ; }
        set { ... = value ; }
    }
}
```

Klassen, Vererbung

```
class A { public          void P() { } ;
          public          void Q() { } ;
          public virtual  void R() { } ;
          public virtual  void S() { } ;
          public virtual  void T() { } ;
          public virtual  void U() { } ; }
class B : A {
          public          void P() { } ;
          public new      void Q() { } ;
          public          void R() { } ;
          public virtuel  void S() { } ;
          public override void T() { } ;
```

Interfaces

Delegates

- sind (Verweise auf) Unterprogramme.
- sind “Bürger (d. h. Daten) erster Klasse”: können auftreten als
  - Werte von Variablen
  - Teile anderer Datentrukturen
  - Argumente und
  - Resultate von Unterprogrammen
- jede Variable eines Delegate-Typs kann eine Liste von Delegates aufnehmen, diese werden bei Benutzung der Reihe nach aufgerufen, der Wert des letzten wird zurückgeliefert.

Remote Procedure Calls

XML-RPC

- ein Client in Java

```
import org.apache.xmlrpc.*;
XmlRpcClientLite c =
  new XmlRpcClientLite
    ("http://dfa.imn.htwk-leipzig.de/cgi-b
Object s = c.execute("examples.add",
  new Vector<Integer>
    (Arrays.asList (new Integer[] { 3, 4 }
```

(vgl. `http://ws.apache.org/xmlrpc/xmlrpc2/`)

- Aufgabe: Protokoll ansehen: Anfrage umleiten, benutze `netcat -l -p 9876` auf lokalem

Autotool-Connector

eigentlich sollte das gehen:

```
server = "http://autotool.imn.htwk-leipzig.de

actor = Actor { schule   = "HTWK Leipzig"
              , matrikel = "NNNNNN"
              , passwort = "XXXXXXX"
              }

problem = Problem { vorlesung = "Objektorient
                  , aufgabe   = "Faktor-Quiz-
                  }
```