Algorithmen & Datenstrukturen I

WS 2002/03

Prof. Dr. Stefan Fischer

4. Abstrakte Datentypen und Objektorientierung

Inhalt

4.1 Einführung

Zentrales Problem bei der Systementwicklung:

Bewältigung von Komplexität

Allgemeines Lösungsprinzip:

1. Rekursives Aufteilen des Gesamtproblems in kleinere möglichst unabhängig zu lösende Teilprobleme.
2. Lösung der Teilprobleme löst dann das Gesamtproblem.

Anwendung dieses Prinzips auf Anweisungsebene:

- Entwicklung von (wiederverwendbaren) Prozeduren und Funktionen

Anwendung dieses Prinzips auf Datenebene:

- Entwicklung von (wiederverwendbaren) Datentypen, die Datenelemente und Operationen *kapseln*.

Erinnerung (Kapitel 3): Grundlegende Datentypen

Definition Datentyp:

Unter einem Datentyp versteht man die Zusammenfassung von Wertebereichen und Operationen zu einer Einheit.

Abstrakter Datentyp:

Schwerpunkt liegt auf den Eigenschaften, die Operationen und Wertebereiche besitzen

Konkreter Datentyp

Darstellung in einer Programmiersprache steht im Vordergrund!

Grundlegende (konkrete) Datentypen: bool, int, real, char, ...

Komplexe Datentypen

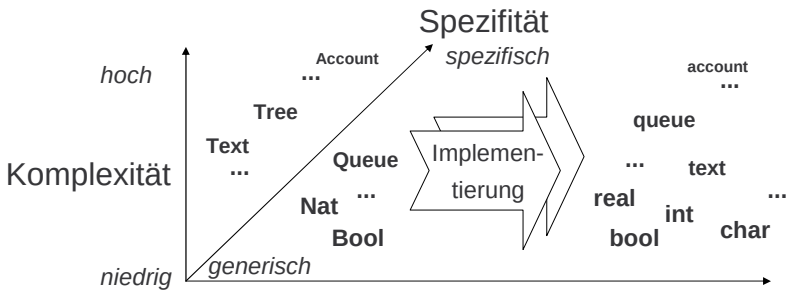
„Komplexe" Datentypen (syn. Datenstrukturen) entstehen i.d.R. durch Kombination anderer Datentypen und besitzen spezifische Operationen. Sie können vorgegeben oder selbstdefiniert sein.

„Atome": Grundlegende Datentypen wie bool, int, float, char etc.

Unterscheidung (in Bezug auf Anwendungsspektrum):

1. *Generische Datentypen*:
   Werden für eine große Gruppe „ähnlicher" Problemstellungen entworfen und sind oft im „Sprachumfang" enthalten (Liste, Keller, Array, Verzeichnis etc.)
2. *Spezifische Datentypen*:
   Dienen der Lösung einer eng umschriebenen Problemstellung und werden i.d.R. im Zusammenhang mit einem konkreten Problem definiert (Datentyp Adresse, Person, Krankenschein etc.)

Datentypen: Klassifizierung



Abstrakte Datentypen >>>>>>> *Konkrete Datentypen*
(Unabhängig von einer konkreten Programmier-sprache) (Abhängig von einer Programmiersprache)

Entwurfsrichtung:

Entwurfsprinzipien

Wünschenswert bei der Spezifikation von Datentypen:

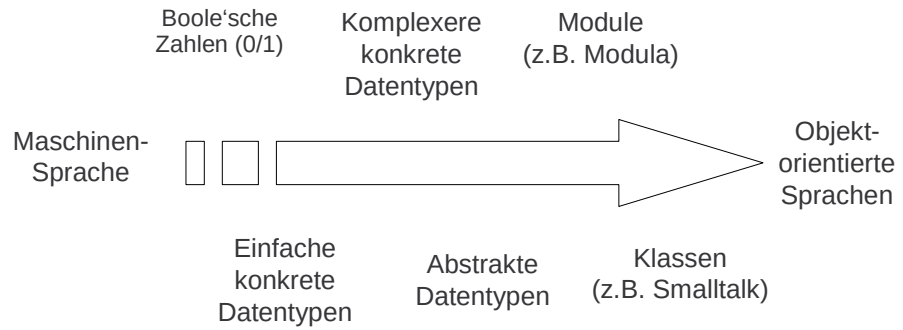
1. Beschreibung (Spezifikation) von Datentypen unabhängig von ihrer späteren Implementierung
   > Spezifikation kann für verschiedene Implementierungen verwendet werden
2. Reduzierung der von außen „sichtbaren" (und somit zugänglichen) Aspekte einer Datentypen auf *ihre Schnittstelle* (= alle von anderen Programmteilen zugänglichen Operationen)
   > Implementierung kann verändert werden, ohne dass die Programmteile verändert werden müssen, die die Struktur benutzen (Wartbarkeit von Programmen)

Entwurfsprinzipien:

1. **(Ein-)Kapselung** (engl. encapsulation): Zugriff nur über Schnittstelle
2. **Geheimnisprinzip** (engl. programming by contract): Die interne Realisierung der Struktur ist verborgen (nur die Schnittstelle ist sichtbar)

„Historische Entwicklung“

Sehr grob:



Entwicklung folgte „dem Bedürfnis“ der Anwender nach immer komplexeren Programmen bzw. „dem Bedürfnis“ der Informatiker, diese beherrschbar zu halten (Stichwort: Softwarekrise)

4.2 Abstrakte Datentypen

Definition: Abstrakter Datentyp (ADT)

Software-Modul mit Operationen $op_1, \ldots, op_n$ als Schnittstelle.

Anmerkungen:

1. Man verwendet einen ADT, *ohne zu wissen*, wie er intern realisiert ist (Geheimnisprinzip) *ausschließlich* über dessen *Schnittstelle* (Kapselung). Insbesondere sind die eigentlichen Datenelemente (i.d.R.) *nicht direkt* zugänglich.
2. Die Funktionalität muss daher *unabhängig* von der internen Realisierung spezifiziert sein.
3. Anhand dieser Spezifikation kann man *verifizieren*, ob ein ADT korrekt realisiert ist.

=> Wichtig ist also die **Spezifikation der Schnittstellen** von ADTs.

Spezifikation von ADTs I (1/2)

Wichtiges Merkmal einer Spezifikation: *Eindeutigkeit*

Daher notwendig: *mathematisch exakte Beschreibungen der Schnittstelle eines ADT und des Modellbegriffs für ADT*

Hier: *Algebraische Spezifikation von ADTs mittels Gleichungen (d.h. Modell eines ADT ist Algebra):*

1. *Signatur* bildet formale Schnittstelle eines ADT
2. Das mathematische Konzept der *(mehrsortigen) Algebra* dient der *Modellbildung* derartig charakterisierter ADTs (d.h. Algebra stellt *Implementierung* des ADT dar)
3. Zu einer Signatur gibt es viele mögliche *Algebren* als Modelle. Daher erfolgt die weitere Spezifikation durch Angabe logischer *Axiome*, die bestimmte Modelle ausschließen (Axiome müssen im Modell gelten) Kernproblem: Eine (die gewünschte) Algebra eindeutig erzwingen!

Achtung: *Themenkomplex der Algebren wird hier nur sehr oberflächlich behandelt (Literatur: z.B. Goos I)*

Spezifikation von ADTs I (2/2)

Eine (ADT-Schnittstellen-) Spezifikation besteht aus einer Signatur $\Sigma = (S,\Omega)$:

In ADT-Spezifikationen ist i.d.R. *eine Sorte* als *neu einzuführende* Sorte gekennzeichnet, die anderen werden *importiert*.

- S Menge von **Sorten**,
- $\Omega = \{ f_{S^*,S} \}$
  Menge von Operatoren (Funktionen) $f : s_1 \ldots s_n \rightarrow s$, jeder bestehend aus einem *Namen f*, einer Folge $s_1,\ldots,s_n \in S$ von *Argumentsorten* und einer *Wertsorte* $s \in S$.
  Funktionen ohne Parameter heißen (auch hier) *Konstante*.

Definition Algebra A_Σ:

Eine Algebra A_Σ zu einer Signatur Σ ist definiert als $A_\Sigma = (A_S, A_\Omega)$ wobei folgendes gilt:

- A_S sind die Trägermengen der Sorten in S
- $A_\Omega = \{A_f : A_{s_1} \ldots A_{s_n} \rightarrow A_s\}$ sind Funktionen auf diesen Trägermengen

Spezifikation und Algebra

Sei nun eine Spezifikation gegeben: wie sieht dann die Klasse der Modelle aus, die zu dieser Spezifikation passt?

Beispiel: ADT Bool (vorläufig)

$S = \{ \text{Bool} \}$

$\Omega = \{ \text{true:} \rightarrow \text{Bool},$
$\text{false:} \rightarrow \text{Bool} \}$

Mögliche Modelle (Algebren) dieser Spezifikation:

1. $A_{Bool} = \{ T, F \}$, $A_{true} := T$; $A_{false} := F$ > *„Erwartungskonform“*
2. $A_{Bool} = \text{IN}$, $A_{true} := 1$; $A_{false} := 0$ > Trägermenge zu groß
3. $A_{Bool} = \{1\}$, $A_{true} := 1$; $A_{false} := 1$ > Passt, aber „ungünstig“

=> Frage: Wie kommt man zu genau der „erwünschten“ Algebra?

Spezifikation von ADTs II

Einschränkung möglicher Algebren durch Angabe von *Axiomen* (gehören zur Spezifikation des ADT):

Definition Axiome Φ :
Menge von Gleichungen, die die Operatoren in ihrer Wirkung beschreiben,

z.B. Funktionsdefinitionen oder auch "nicht ausführbare" (deskriptive) Spezifikationen

Beispiel für eine „nicht ausführbare" Spezifikation:

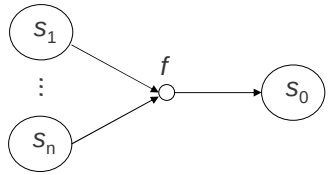
Die Wurzelfunktion $sqrt(x) = \sqrt{x}$ auf den nichtnegativen reellen Zahlen IR^+ lässt sich spezifizieren durch

$$\text{sqrt}(x) \geq 0 \wedge \text{sqrt}(x) * \text{sqrt}(x) = x \ .$$

Dies definiert die Funktion eindeutig, gibt aber keinen Hinweis darauf, wie man sie berechnen kann.

Signaturdiagramme

Signaturen lassen sich übersichtlich durch **Signatur-Diagramme** mit Sorten als Knoten und Operatoren als (Hyper-) Kanten darstellen:



Notation für Operatornamen

Mit einem Platzhaltersymbol '_' für Argumente kann man folgende Notationen für Operatornamen einführen:

	Operatoren	Beispiele
Infix	_≤_, _+_, _∨_, ...	$a \leq b$, $n+m$, $p \vee q$, ...
Postfix	_!, _², ...	$n!$, x^2, ...
Mixfix	|_|, |_-_|, if_then_else_fi, ...	$\|x\|$, $\|x-y\|$, ...

Bei Standard-Präfixnotation schreiben wir *f* statt *f* (_,...,_).

Spezifikation der grundlegenden Datentypen (1/4)

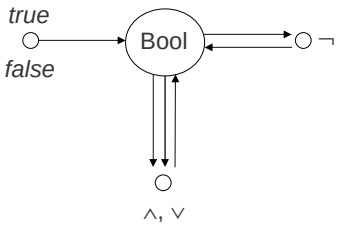
Der ADT der Boole'schen Werte:

$S = \{$ Bool $\}$

$\Omega = \{$ *true*: $\rightarrow$ Bool,
false: $\rightarrow$ Bool,
$\neg$_: Bool $\rightarrow$ Bool,
$\vee$: Bool$\times$Bool $\rightarrow$ Bool,
_ $\wedge$_: Bool$\times$Bool $\rightarrow$ Bool, ...$\}$

$\Phi = \{$ *false* $\wedge$ *x* = *false*,
x $\wedge$ false = *false*,
true $\wedge$ *true* = *true*;
true $\vee$ *x* = *true*,
x $\vee$ *true* = *true*,
false $\vee$ *false* = *false*,
$\neg$ *false* = *true*, $\neg$ *true* = *false* $\}$

Signaturdiagramm:



Spezifikation der grundlegenden Datentypen (2/4)

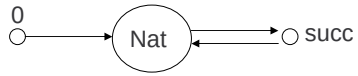
Der ADT der natürlichen Zahlen I/II:

$S = \{$ Nat $\}$

$\Omega = \{$ 0: $\rightarrow$ Nat,
succ: Nat $\rightarrow$ Nat $\}$

$\Phi = \{ \}$

Signaturdiagramm:



Anmerkungen:

- Damit wird z.B. 3 als succ(succ(succ(0))) = $\text{succ}^3(0)$ dargestellt.
- Der Term $\text{succ}^n(0)$ stellt die natürliche Zahl *n* dar und lässt sich nicht weiter umformen: es gibt keine Gleichungen.

Spezifikation der grundlegenden Datentypen (3/4)

Der ADT der natürlichen Zahlen II/II:

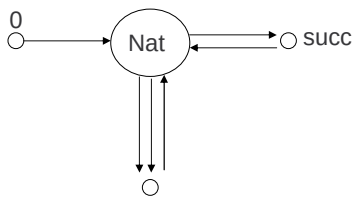
Erweiterung von Nat *um die Addition:*

$S = \{$ Nat $\}$

$\Omega = \{$ 0: $\rightarrow$ Nat,
succ: Nat $\rightarrow$ Nat,
add: Nat$\times$Nat $\rightarrow$ Nat $\}$

$\Phi = \{$ add(x,0) = x,
add (x, succ(y))
= succ(add(x,y)) $\}$

Signaturdiagramm:



Anmerkungen:

- add (x, succ(y)) = succ(add(x,y)) entspricht $x+(y+1) = (x+y)+1$
- Dies ist eine abstrakte Spezifikation. Reale Implementierungen nutzen natürlich die Hardware-Integers, obwohl diese nicht streng obigen Gleichungen genügen, da die Zahlendarstellung beschränkt ist.

Spezifikation der grundlegenden Datentypen (4/4)

Beispiel zur Anwendung der Gleichungen:

"2+3"= add(succ(succ(0)),succ(succ(succ(0))))
② = succ(add(succ(succ(0)),succ(succ(0))))
② = succ(succ(add(succ(succ(0)),succ(0))))
② = succ(succ(succ(add(succ(succ(0)),0))))
① = succ(succ(succ(succ(succ(0)))))

$S = \{$ Nat $\}$
$\Omega = \{$ 0: $\rightarrow$ Nat,
succ: Nat $\rightarrow$ Nat,
add: Nat$\times$Nat $\rightarrow$ Nat $\}$
$\Phi = \{$ add(x,0) = x, ①
add (x, succ(y)) ②
= succ(add(x,y)) $\}$

Anmerkung:
Der ADT **Nat** hat eine schöne Eigenschaft: jeder Term hat eine eindeutige *Normalform* succ^n**(0)**, der n darstellt. Diese entsteht, wenn man die Gleichungen nur von links nach rechts anwendet, bis alle add-Operationen verschwunden (="ausgeführt") sind.

Implementierung (1/2)

ADT *S* ⟹ Implementierung ⟹ ***Algebra A (im Rechner repräsentierbar)***

Implementierung eines ADT heißt:

- Abbildung der Sorte S in „Datenstruktur" $\boldsymbol{A_S}$
  Beispiel: Nat $\rightsquigarrow A_{Nat} = IB^m$ (m-stellige Vektoren über {0,1})
- Abbildung der Operatoren $f : s_1 \ldots s_n \rightarrow s$ in Funktionen $A_f : A_{s_1} \ldots A_{s_n} \rightarrow A_s$
  Beispiel:
  - succ: Nat $\rightarrow$ Nat $\rightsquigarrow$ _+1: $IB^m \rightarrow Ib^m$
  - add: Nat $\times$ Nat $\rightarrow$ Nat $\rightsquigarrow$ _+_: $IB^m \times IB^m \rightarrow IB^m$
- Sicherstellen, dass alle Axiome gelten (in den Grenzen der darstellbaren Werte bzw. der Darstellungsgenauigkeit)
  Beispiel: $x+0 = x$ *und* $x+(y+1) = (x+y)+1$

Implementierung (2/2)

Beispiel ($A_{Nat} = IB^m$):

Darstellung von $x \in$ IN mit m Binärziffern $z_{m-1},\ldots,z_0 \in$ IB:

$$x = \sum_{i=0}^{m-1} z_i \cdot 2^i$$

Darstellbarer Zahlbereich: $0 \leq x \leq 2^{m-1}$

Beispiel: 42 ≅ 101010 wird auch als LOLOLO geschrieben

Addition von Binärzahlen: Ziffernweise mit Übertrag:
O+O=O, O+L=L+O=L, L+L=LO

Beispiel:	**0**	**L**	**0**	**L**	**0**	**L**	**0**
	+ 0	**0**	**L**	**0**	**L**	**L**	**L**
	L	**0**	**0**	**0**	**0**	**0**	**L**

Die Gültigkeit der Rechengesetze muss überprüft werden !

Alternative Notation

Im Folgenden wird alternativ zur mathematischen Schreibweise zur Spezifikation von ADTs folgende an Programmiersprachen angelehnte Notation genutzt (am Beispiel des ADT Nat):

Schreibweise 1:

S = { Nat }

Ω = { 0: → Nat,
succ: Nat → Nat,
add: Nat×Nat → Nat }

Φ = { add(x,0) = x,
add (x, succ(y))
= succ(add(x,y)) }

Schreibweise 2:

type Nat

import ∅

operators

0: → Nat
succ: Nat → Nat
add: Nat×Nat → Nat

axioms ∀ *i,j*: Nat
add (*i*,0) = *i*
add (*i*, succ(*j*))
= succ (add (*i*, *j*))

Die import-Anweisung (bei **Nat** ohne Angabe) erlaubt die Angabe der Sorten, die zusätzlich zur zu definierenden Sorte (über type benannt) benötigt werden.

6.3 Beispiele für Abstrakte Datentypen

1. **Parametrisierte Datentypen**
2. **Lineare Datentypen**
3. **Listen**
4. **Keller bzw. Stapel (Stack)**
5. **Warteschlangen (Queues)**
6. **Mengen**

Parametrisierte Datentypen

Die im Folgenden behandelten ADTs sind *parametrisiert*:

Der Typ der Elemente (i.d.R. mit T bezeichnet) tritt als Parameter (genauer *Sortenparameter*) auf und kann verschieden instanziiert werden, z.B. **List**(T) als **List**(**Bool**), **List**(**Nat**), **List**(**List**(**List**(**Nat**))),...
Achtung: Der Sortenparameter wird nicht importiert!

Lineare Datenstrukturen

Die im Folgenden behandelten ADTs *Liste*, *Keller* und *Warteschlange* sind *linear*: Ihre Implementierung kann auf lineare Listen zurückgeführt werden, d.h. Listen ohne Unterlisten.

Nichtlineare ADTs sind:

– Bäume,

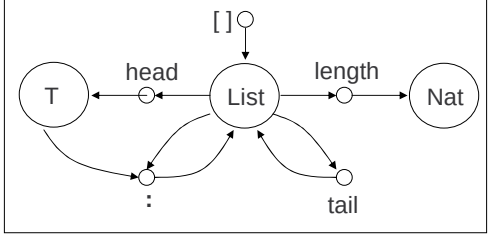
– Graphen etc.

und werden in späteren Abschnitten behandelt.

Listen

Hier eine ADT-Spezifikation mit einer kleinen Auswahl der Listenoperationen.

Signaturdiagramm:



Listen dienen als Implementierungsgrundlage für die nachfolgenden linearen ADTs; für Listen selbst geben wir keine Implementierung an: sie sind „fest eingebaut“.

type List(T)

import Nat

operators

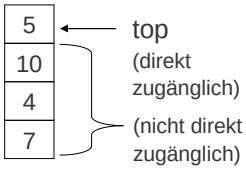
[ ]: → List
_:_ : T×List → List
head : List → T
tail : List → List
length : List → Nat

axioms ∀ *l*: List, ∀ *x*: T
head (*x*:*l*) = *x*
tail (*x*:*l*) = *l*
length([])=0
length(*x*:*l*)
= succ(length(*l*))

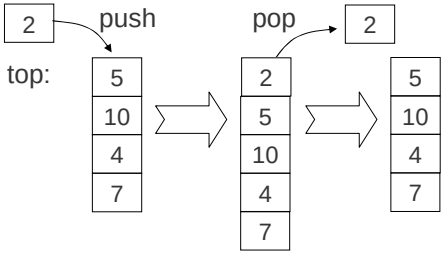
Keller bzw. Stapel (Stack) 1/2

. . . sind Listen, bei denen nur auf das ”oberste“ Element zugegriffen werden kann. LIFO-Speicher: *Last In-First Out; die letzten werden die ersten sein*.

Aufbau:



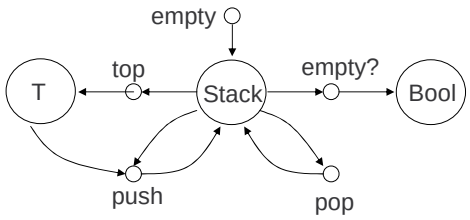
Hinzufügen und Entfernen eines Elements:



Keller bzw. Stapel (Stack) 2/2

type Stack(T)
import Bool
operators
empty: → Stack
push: Stack×T → Stack
pop : Stack → Stack
top : Stack → T
empty?: Stack → Bool
axioms ∀ *s*: Stack, ∀ *x*: T
pop(push(*s*,*x*)) = *s*
top(push(*s*,*x*)) = *x*
empty?(empty)= true
empty?(push(*s*,*x*)) = false

Signaturdiagramm:



Anmerkung:
pop(empty) = ⊥
top(empty) = ⊥
Diese Fälle bleiben undefiniert!

Keller: Implementierung

Übliches Verfahren bei der Implementierung von Datenstrukturen: Rückgriff auf bereits vorhandene Strukturen (>Wiederverwendung).

Idee: Realisierung (Implementierung) des ADT Keller durch Listen

- Die Sorte Stack (genauer: Stack(T)) wird implementiert durch die Menge $A_{List(T)}$ der Listen über T.
- Die Operatoren werden durch die folgenden gleichnamigen Funktionen implementiert:

empty	= [ ]
push(*l*,*x*)	= *x* : *l*
pop(*x* : *l*)	= *l*
top(*x* : *l*)	= *x*
empty?([ ])	= *true*
empty?(*x*:*l*)	= *false*

Fehlerfälle:
pop(empty) = ⊥
top(empty) = ⊥
Hier bleiben diese Fehler unbehandelt. In einer tatsächlichen Implementierung müssen hierfür (natürlich) Vorkehrungen getroffen werden!

Keller: Verwendung

In der Mathematik: Auswertung von Termen, die aufgrund von Klammerungen oder Rechengesetzen nicht sofort ausgerechnet werden können

Beispiel: (3+5*7)-(5+3) = ?

Wenn man den Term von links nach rechts berechnen würde, würde das Prinzip „Multiplikation vor Addition“ verletzt. Also muss man die gelesenen Daten zwischenspeichern, bis man sie benötigt.

Andere wichtige Verwendung: Ablegen von Parametern einer Prozedur/Funktion, wenn innerhalb eine weitere Funktion aufgerufen wird.

Damit stehen diese Daten später wieder zur Verfügung, wenn die innere Funktion abgearbeitet ist.

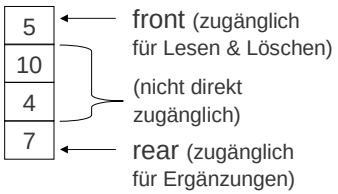
Wird in praktisch allen Betriebssystemen/Programmiersprachen eingesetzt.

Warteschlangen (1/2)

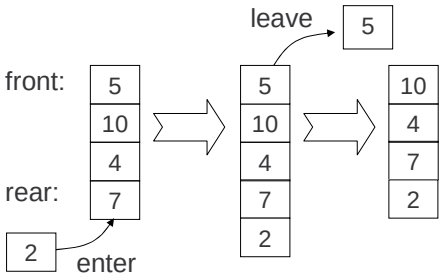
. . . sind Listen, bei denen nur an *einem* Ende eingefügt und nur am *anderen* Ende gelöscht werden kann. Lesbar ist nur das zu löschende Element.

FIFO-Speicher: *First In-First Out, wer zuerst kommt, mahlt zuerst*.

Aufbau:



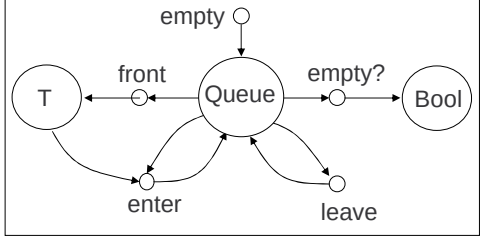
Hinzufügen und Entfernen eines Elements:



Warteschlangen (2/2)

type Queue(T)
import Bool
operators
empty: → Queue
enter: Queue ×T → Queue
leave: Queue → Queue
front: Queue → T
empty?: Queue → Bool
axioms ∀ *q*: Queue, ∀ *x*: T
leave(enter(empty,*x*)) = empty
leave(enter(enter(*q*,*x*), *y*))
= enter(leave(enter(*q*,*x*)), *y*)
front(enter(empty,*x*)) = *x*
front(enter(enter(*q*,*x*),*y*))
= front(enter(*q*,*x*))
empty?(empty) = true
empty? (enter(*q*,*x*)) = false

Signaturdiagramm:



Anmerkung:
front(empty) = ⊥
leave(empty) = ⊥
Diese Fälle bleiben undefiniert!

Warteschlangen: Implementierung

Idee: Realisierung (Implementierung) des ADT Queue durch Listen

- Die Sorte Queue (genauer: Queue(T)) wird implementiert durch die Menge $A_{List(T)}$ der Listen über T (Kopf der Liste ≅ rear).
- Die Operatoren werden durch die folgenden gleichnamigen Funktionen implementiert:

empty = [ ]
enter(*l*,*x*) = *x* : *l*
leave(*x* : []) = []
leave(*x* : *l*) = *x* : leave(*l*)
front(*x* : []) = *x*
front(*x* : *l*) = front(*l*)
empty?([]) = *true*
empty?(*x*:*l*) = *false*

Fehlerfälle:
leave(empty) = ⊥
front(empty) = ⊥
Hier bleiben diese Fehler unbehandelt. In einer tatsächlichen Implementierung müssen hierfür (natürlich) Vorkehrungen getroffen werden!

Warteschlangen: Verwendung

Eine häufige Anwendung sind Algorithmen zur Vergabe von Ressourcen an Verbraucher (z.B. Betriebssystem)

- Prozessverwaltung
  - Ressource: Rechenzeit
  - Elemente der Warteschlange: rechenwillige Prozesse
  - Grundidee: Jeder Prozess darf eine feste Zeit lang rechnen, wird dann unterbrochen und hinten in der Warteschlange wieder eingereiht (sofern noch Bedarf an Rechenzeit vorhanden)
- Druckerverwaltung (Drucker-Spooler)
  - Ressource: Drucker
  - Elemente der Warteschlange: Druckaufträge
  - Grundidee: Druckaufträge werden bzgl. Zeitpunkt ihres Eintreffens nacheinander vollständig abgearbeitet
- Speicherverwaltung etc.

Allgemein: Koordinierung asynchroner Prozesse

Mengen (1/2)

ADT mit vielen Anwendungen und Implementierungsmöglichkeiten.

Beispiele : Relationale Datenbanken; Verzeichnisse aller Art.

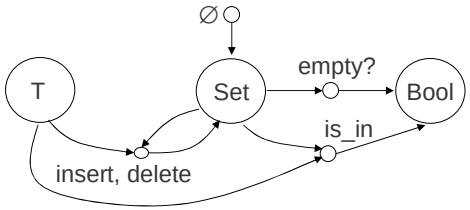
Gegeben: Elementtyp T mit Gleichheitsrelation =

Hier: Sehr einfache Spezifikation mit nur wenigen Operatoren

Achtung: Unterscheidung von Menge (Set) und Mehrfachmenge (Bag)

Hier: nur *ein* Vorkommen eines Elements (Set)

Signaturdiagramm:



Mengen (2/2)

Spezifkation:

type Set(T)
import Bool
operators
∅ : → Set
empty? : Set → Bool
insert : Set ×T→ Set
delete : Set ×T→ Set
is_in : Set ×T→ Bool
axioms ∀ *s*: Set, ∀ *x,y*: T
empty?(∅) = true
empty?(insert(*s,x*)) = false
is_in(∅,*x*) = false
is_in(insert(*s*,*x*), *y*) =
if *x*=*y* **then** true **else** is_in(*s,y*) **fi**

axioms (Fortsetzung)
insert(insert(s,x),y) =
if x=y **then** insert(s,x)
else insert(insert(s,y),x) **fi**
delete(∅,*x*) = ∅
delete(insert(s,*y*), *x*) =
if *x*=*y* **then** delete(*s,x*)
else insert(delete(*s,x*), y) **fi**

Anmerkung: Folgendes gilt aufgrund der bekannten Mengensemantik:

- insert(insert(s,x),y)
  = insert(insert(s,y),x)

Diese Regel kann mit den Gleichungen aber nicht bewiesen werden! Einführung von Axiomen zur Gleichsetzung?

Mengen: Implementierung

Idee: Realisierung (Implementierung) des ADT Set durch Listen

- Die Sorte Set (genauer: Set(T)) wird implementiert durch die Menge $A_{List(T)}$ der Listen über T.
- Die Operatoren werden durch die folgenden gleichnamigen Funktionen implementiert:

∅ = [ ]
is_in (*x*, []) = false
is_in (*x*, *x* : *l*) = true
is_in (*x*, *y* : *l*) = is_in (*x*, *l*)
insert(*x, l*) =
if is_in(*x*,*l*) **then** *l* **else** *x*:*l* **fi**
empty?([]) = *true*
empty?(*x*:*l*) = *false*

delete (*x*, []) = []
delete (*x*, *x* : *l*) = *l*
delete (*x*, *y* : *l*) =
insert(delete(x,*l*),y)

Fehlerfälle:
delete(empty) = ⊥

4.4 Entwurf von Datentypen (1/2)

Der Entwurf von ADT ist nicht trivial!
Ausgehend von der „Idee“ eines ADT für eine bestimmte Problemklasse stellen sich u.a. folgende Fragen:

1. Wie komme ich zu den Gleichungen?
2. Wann habe ich genug Gleichungen?
3. Wann fehlt mir ein Axiom?

Vorgehensweise:

1. Festlegung der *Konstruktorfunktionen*
   Bestimmt den Bereich der Basisterme, die den später implementierten Werten entsprechen (z.B. empty und push)
2. Ggf. Axiome zur *Gleichsetzung von Konstruktortermen* einführen (z.B. bei ADT Set)
3. Definition geeigneter *Selektorfunktionen*
   Dienen dem lesenden Zugriff, ohne neue Werte zu konstruieren (z.B. front oder top)

4.4 Entwurf von Datentypen (2/2)

Vorgehensweise (Fortsetzung):

4. Definition geeigneter *Manipulatoren*
   Dienen der Manipulation der Datenwerte und haben als Ergebniswert den neu spezifizierten Datentyp (z.B.: pop).
   Wichtig: vollständige Fallunterscheidung.
5. Abfangen von *Fehlersituationen*

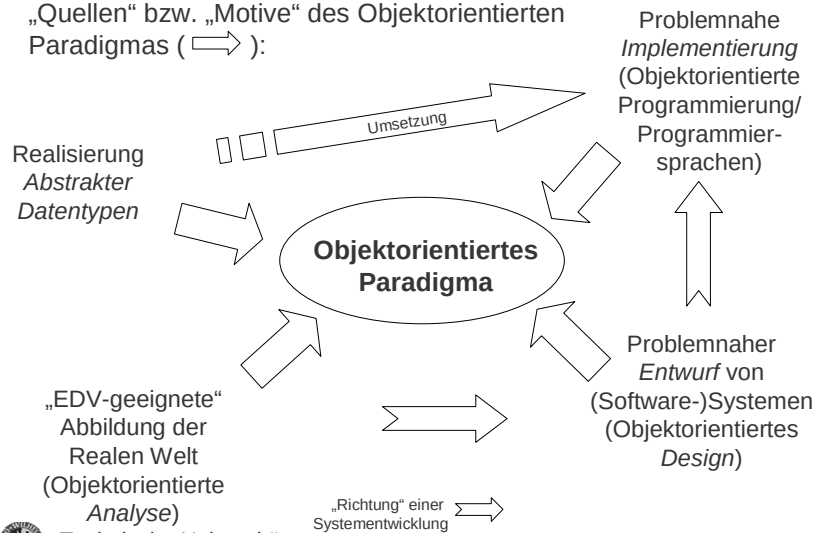
Bei der Spezifikation von Manipulatoren:

- Nach Möglichkeit auf bereits spezifizierte Manipulatoren zurückgreifen.
- Aufbau der Regeln von links nach rechts als Ersetzungsregeln anstelle der Formulierung beliebiger Gleichungen.
  Wichtig hierbei: die rechte Seite sollte einfacher als die linke Seite sein (z.B. durch „Schrumpfen“ eines Parameterwertes).
- Analog bei rekursiv definierten Funktionen: die Argumente des rekursiven Aufrufs sollten einfacher werden und natürlich muss der Abbruch garantiert sein.

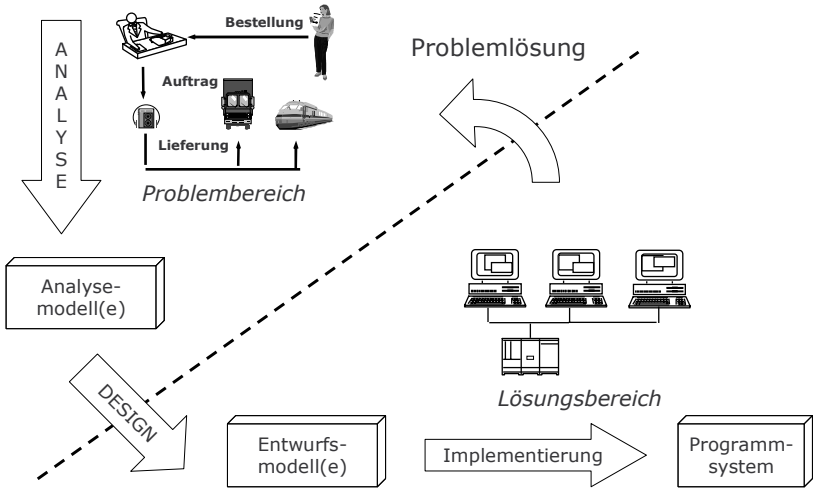
4.5 Objektorientierung

Einführung

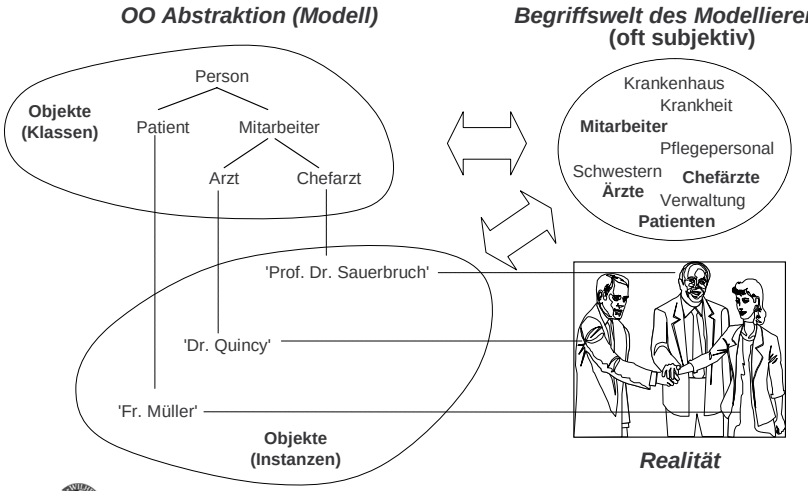
„Quellen“ bzw. „Motive“ des Objektorientierten Paradigmas (⇨):



Erinnerung: Softwareentwicklung



Grundlagen der Objektorientierung (1/10)



Grundlagen der Objektorientierung (2/10)

Definition Objekt:

Ein *Objekt* ist die Repräsentation eines Gegenstandes oder Sachverhalts der realen Welt oder eines rein gedanklichen Konzepts. Es ist gekennzeichnet durch:

1. eine eindeutige *Identität*, durch die es sich von anderen Objekten unterscheidet
2. statische Eigenschaften zur Darstellung des *Zustandes* des Objekts in Form von *Attributen*
3. dynamische Eigenschaften in Form von *Methoden*, die das *Verhalten* des Objekts beschreiben.

Grundlagen der Objektorientierung (3/10)

Beispiele:

1. *Person p* mit *Namen* Müller und *Geburtsdatum* 24.12.22 (*Attribute mit Belegung* = *Zustand*) und *Methode alter*():Integer
2. *Rationale Zahl r* mit *Zähler z* und *Nenner n (Attribute)* und *Methoden normalisiere*(), *addiere*(r: RationaleZahl)
3. *Leerer Stack s* mit *Methoden pop*():Objekt, *push*(t) ...

Anmerkungen:

- Der Zustand (zu einem Zeitpunkt) eines Objekts entspricht der Belegung der Attribute des Objekts (zu diesem Zeitpunkt).
- Methoden entsprechen in der programmiersprachlichen Umsetzung Prozeduren bzw. Funktionen, denen Parameter übergeben werden können. Der Zustand eines eine Methode ausführenden Objekts (und nur der dieses Objekts) ist den Methoden im Sinne einer Menge globaler Variablen direkt zugänglich (d.h. kann gelesen und verändert werden).

Grundlagen der Objektorientierung (4/10)

Objektmodelle:

- *Wertbasierte Objektmodelle*

  Diese Modelle beinhalten keine Objektidentität i.e.S., da die Einzigartigkeit eines Objektes *auf dem Objektzustand* basiert.
  (Zwei Objekte sind dann *gleich* bzw. *dieselben*, wenn sie den gleichen *Zustand* haben; Beispiel: Person p_1 mit Namen „Peter Müller“ (einziges Attribut) und Person p_2 mit Namen „Peter Müller“ $\Rightarrow p_1 = p_2$)

- *Identitätsbasierte Objektmodelle*

  Jedem Objekt innerhalb des Systems wird eine vom Wert unabhängige Identität zugeordnet (im Beispiel oben *können* p_1 und p_2 unterschiedliche Objekte bezeichnen, d.h. $p_1 \neq p_2$).

- *Hybride Objektmodelle*

  In diesen Modellen kann der Programmierer/Modellierer spezifizieren, ob ein Attribut ein Wert oder ein Objekt mit Identität enthalten soll.

Grundlagen der Objektorientierung (5/10)

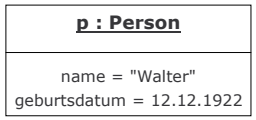
Grafische Notation:

Im Folgenden wird zur grafischen Notation von Objektsystemen die *Unified Modeling Language (UML)* verwendet (allerdings vereinfacht)!

>„De Facto“-Standard zur Modellierung von Objektsystemen bei *Analyse* und *Entwurf* von (objektorientierten) Softwaresystemen.

Notation eines Objekts:

Beispiel:



Grundlagen der Objektorientierung (6/10)

Zentrale Eigenschaft von Objekten:

Objekte *kapseln* analog zu Abstrakten Datentypen ihre „Interna“ (> *Geheimnisprinzip*), d.h.

- ihren Zustand (= Belegung aller Attribute des Objekts)
- die „Implementierung“
  - ihrer Zustandsrepräsentation
  - ihres Verhaltens (d.h. ihrer Methoden)

Objekte sind *einzig über ihre Schnittstelle*, d.h. über die Menge der vom Objekt der „Außenwelt“ des Objekts zur Verwendung zur Verfügung gestellten Methoden (syn. *Dienste*) zugänglich!

Anmerkung: es gibt sog. „private“ Methoden, die nur dem Objekt selbst (innerhalb der eigenen Methoden) zugänglich sind (vereinfacht)

Grundlagen der Objektorientierung (7/10)

Objekte interagieren über *Nachrichten*:

- Ein Objekt *x versendet* eine *Nachricht n* an Objekt *y*.
- Mit dieser Nachricht *n* verbunden ist die Auswahl einer Methode *m* der Schnittstelle des Objekts y.
- Übergeben werden entsprechend der Spezifikation der Methode Parameter.
- Zurückgegeben wird (insofern die Methode als Funktion definiert ist) ein Rückgabewert.

Anmerkungen:

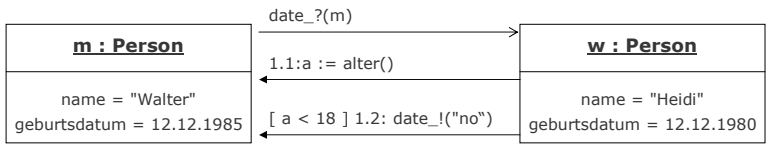
- Methodenaufrufe können den Zustand eines Objekts verändern
- Ein Objekt kann sich selbst eine Nachricht schicken (und so auch private Methoden aufrufen)
- Nachrichten entsprechen i.d.R. Prozedur-/Funktionsaufrufen im Programmtext einer „aufrufenden“ Methode.

Grundlagen der Objektorientierung (8/10)

Notation *Nachrichten (vereinfachtes Kollaborationsdiagramm)*:

Pfeil mit Beschriftung:

1. ggf. Bedingung in eckigen Klammern
2. (Hierarchische) Nummerierung zur Kennzeichnung der Reihenfolge der Nachrichten
3. ggf. Zuweisung
4. (:) Methodenname (mit Parametern)



→ synchron (Übergebe Kontrollfluss > Warten auf Rückgabe)
→ asynchron (Aufteilen des Kontrollflusses > nur Prozedur)

Grundlagen der Objektorientierung (9/10)

Objekte können *in Beziehung* zueinander stehen.

Beispiel: Firma *f* steht in Beziehung „Beschäftigungsverhältnis“ zu Person *p*

Die Beteiligten an einer Beziehung nehmen bezüglich der Beziehung *jeweils eine Rolle* ein!

Beispiel: Rolle von Firma *f*: „Arbeitgeber“,
Rolle von *p*: „Arbeitnehmer“

Ein Objekt kann *mit mehreren* Objekten in der gleichen Beziehung stehen.

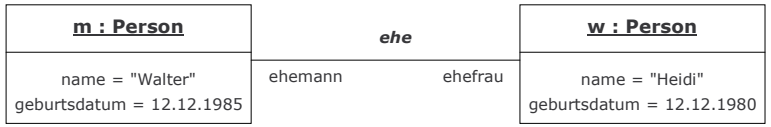
Beispiel: Firma *f* steht in Beziehung „Beschäftigungsverhältnis“ zu Person p_1 *und* Person p_2

Grundlagen der Objektorientierung (10/10)

Notation *Beziehung (vereinfacht)*:

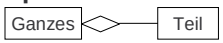
Linie mit Beschriftung:

1. Bezeichnung der Beziehung kursiv in der Mitte
2. Rollenbezeichnungen nahe dem jeweils zugehörigen Objekt

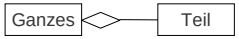


Spezialfall einer Beziehung: Aggregation und Komposition

Aggregation: Modellierung von Teil-Ganzes-Beziehungen



Komposition: Aggregation mit existenzieller Bindung (ohne Teil kein Ganzes)



Klassen und Objekte (1/17)

Bisher: Objekte verfügen über Attribute, Verhalten und eine Identität, gehen Beziehungen zu anderen Objekten ein und interagieren über Nachrichten.

Beobachtung: Es gibt sehr viele Objekte, die bzgl. Attributen, Verhalten und Beziehungen *sehr ähnlich* sind (z.B. Person p_1 und p_2 , Rationale Zahlen 1/3 und 4/5 etc.)

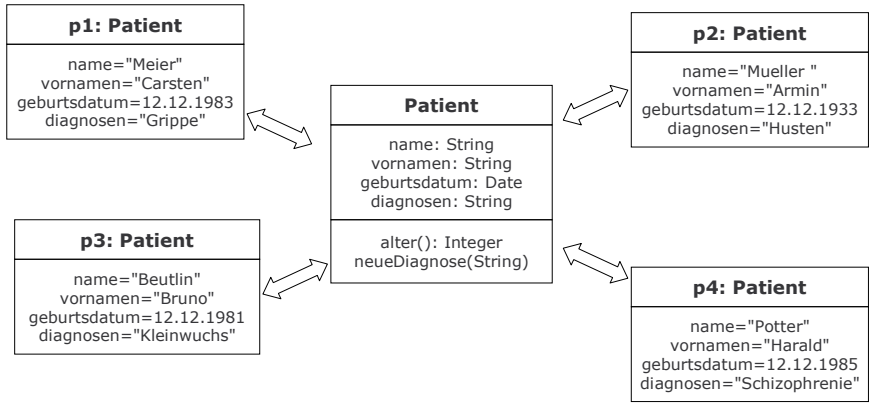
Frage: Sollen für jedes dieser Objekte diese immer gleichen Strukturen *immer wieder neu* angegeben/implementiert werden?

Antwort: NEIN!
Man fasst ähnliche Objekte zu einer **Klasse** zusammen, die Aussagen über die Struktur (Attribute, Beziehungen) und das Verhalten der zugehörigen Objekte zusammenfasst und als „Ansprechpartner“ bzgl. Objekterzeugung etc. dient.

=> Eine Klasse ist i.d.R. ebenfalls ein Objekt (einer sog. Metaklasse)

Klassen und Objekte (2/17)

Klassenbildung: „ähnliche Objekte“ werden zu einer Klasse zusammengefasst!



Redeweise: p1 *ist Instanz* der Klasse Patient

Klassen und Objekte (3/17)

Definition Klasse:

Anschaulich: Eine *Klasse* ist eine Menge von Objekten mit gleichen Eigenschaften und gleichem Verhalten.

Genauer: Eine *Klasse* ist eine Beschreibung von Objekten (ggf. nur einem), die über eine *gleichartige Menge von Attributen und Methoden* verfügen und beinhaltet eine *Beschreibung*, wie neue Objekte dieser Klasse *erzeugt* werden können.

Anmerkungen:

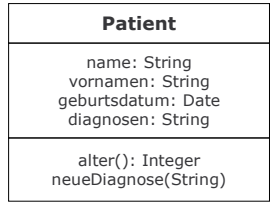
- Eine Klasse(ndefinition) ist eng verwandt mit einem ADT (bzw. dessen Spezifikation): sie legt Attribute und Methoden der dazugehörigen Objekte fest.
- Objekte einer Klasse nennt man auch *Instanzen* dieser Klasse!

Klassen und Objekte (4/17)

Grafische Notation:

Die grafische Notation einer Klasse ist ähnlich der Notation von Objekten.
In einem zweifach unterteilten Rechteck werden notiert:

- Klassenname
- Attribute (Attributname, gefolgt von ':' und der Angabe der Klasse, aus denen die Objekte zu Belegung des Attributs entstammen dürfen)
- Dienste (Deklarationsteil der Methode)



Oft wird als Kurzform nur ein Rechteck mit dem Klassennamen gezeichnet

Exkurs: Java 1/3

Java ist eine imperative objektorientierte Programmiersprache mit identitätsbasiertem Objektmodell.

(Kurze und unvollständige) vergleichende Darstellung der Syntax:

Anweisung	**Imp. Alg.**	**Java**
Zuweisung	:=	=
Sequenz	;	;
Block	do..od, if..fi ...	{...}
Alternative	if *P* then α_1 fi	if (*P*) {α_1}
	if *P* then α_1 else α_2 fi	if (*P*) {α_1} else {α_2}
While-Schleife	while *P* do α od	while (*P*) {α_1}
For-Schleife	for *i* := *j* to *k* do α od	for (*i=j*; *i<=k*; *i++*) {α}

Anmerkung:
Bei Blöcken mit nur einer Anweisung kann die Klammerung entfallen.

Exkurs: Java 2/3

Grunddatentypen	**Imp. Alg.**	**Java**
	int, bool, real ...	int, boolean, float,...
Var.-deklaration	var x,y: int	int x,y
Array	var array *x*[1..*n*]: int	int[] x = new int[*n*]
Zugriff	x[1]:=2	x[0]=2
Anmerkung: in Java beginnt die Indexierung mit 0		
Operatoren		
Test auf Identität		==
Test auf Gleichheit	=	== od. spez. Methode
Ungleichheit	≠	!=
Ordnungsrel.	<, ≤, ≥, >	<, <=, >=, >
Boole'sche Op.	∧, ∨, ¬	&, |, !
		&&, ||

Anmerkung: bei letzten beiden Operatoren wird der jeweils zweite Operand nur ausgewertet, wenn der erste true (&&) bzw. false (II) ergibt)

Exkurs: Java 3/3

Prozeduren und Funktionen der imp. Progr. und Java-Methoden

Syntax der Prozeduren der imp. Progr. (Kap. 4)

proc P([var] p_1: τ_1, ... , [var] p_n: τ_n) {{ δ ; $\alpha(p_1, ... , p_n)$}}

Methoden <u>ohne</u> Rückgabewert in Java (ohne Modifikator)

void P ($\tau_1 p_1$, ..., $\tau_n p_n$) {„Anweisungen & Deklarationen“}

Zur Parameterart (Werteparameter, Referenzparameter etc.):
Java kennt nur bei primitiven Datentypen (int, boolean etc.) und der Klasse String die Möglichkeit des Werteparameters. Ansonsten werden immer Objekte bzw. Objektreferenzen übergeben (Referenzparameter)

Syntax der Funktionen der imp. Progr. (Kap. 4)

fun F([var] p_1: τ_1, ... , [var] p_n: τ_n):τ {{ δ , $\alpha(p_1, ... , p_n)$; $t(p_1, ... , p_n)$ }}

Methoden <u>mit</u> Rückgabewert in Java (ohne Modifikator)

τ P ($\tau_1 p_1$, ..., $\tau_n p_n$) {„Anweisungen & Deklarationen &
ggf. mehrere return-Anweisungen“}

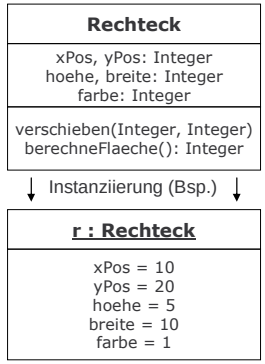
Klassen und Objekte (5/17)

Klassen und Objekte in Java I (1/7):

Beispiel: die Klasse Rechteck
Die Klasse Recheck definiert für ihre Instanzen die Attribute

- xPos, yPos: Gibt die x- und y-Position der linken oberen Ecke des Rechtecks (Ursprung) im 2D-Koordinatensystem an
- hoehe, breite: Angabe zur Höhe und Breite des Rechtecks in Bildpunkten (Pixeln)
- farbe: gibt als Zahl codiert die Farbe des Rechtecks an

und Methoden (Dienste)

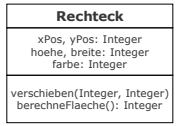


- verschieben: der Ursprung des Rechtecks wird entsprechend der übergeben Integer-Werte verschoben (durch Addition) => Manipulation
- berecheFlaeche: die Fläche des durch hoehe und breite aufgespannten Rechtecks wird berechnet und zurückgegeben

Klassen und Objekte (6/17)

Klassen und Objekte in Java I (2/7):

Als (imperative) objektorientierte Programmiersprache mit identitätsbasiertem Objektmodell unterstützt Java die genannten Konzepte. Die Klasse Rechteck in Java:




```
// Definition der Klasse Rechteck
public class Rechteck {
   // Attributdeklarationen
   int xPos, yPos, hoehe, breite;
   int farbe;
   // Methoden
   // Konstruktoren
   public Rechteck () {
         xPos = yPos = 0;
         breite = hoehe = 10;
   }
```

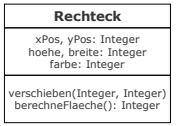
```
// Fortsetzung Konstruktoren
   public Rechteck (int xp, int yp, int b,
                                 int h) {
         xPos = xp;
         yPos = yp;
         breite = b;
         hoehe =h;
   } ...
```


Klassen und Objekte (7/17)

Klassen und Objekte in Java I (3/7):

Die Klasse Rechteck in Java (Fortsetzung):

```
// Fortsetzung Methoden
// Manipulatoren und Selektoren
// Methode zum Verschieben (Manipul.)
   public void verschieben (int dx, int dy)
   {
         xPos = xPos + dx;
         yPos = yPos + dy;
   }
// Methode zur Flächenberechnung (Sel.)
   public int berechneFlaeche () {
         return breite*hoehe;
   }
} // Ende der Klassendefinition
```



Anmerkungen:

- Klassendefinition ist ein durch geschweifte Klammern eingerahmter Block, der mit dem Klassennamen eingeleitet wird (nach dem Schlüsselwort class).
- Attribute einer Klasse entsprechen lokalen Variablen; Lebensdauer = Lebensdauer des Objekts

Klassen und Objekte (8/17)

Klassen und Objekte in Java I (4/7):

Anmerkungen (Fortsetzung):

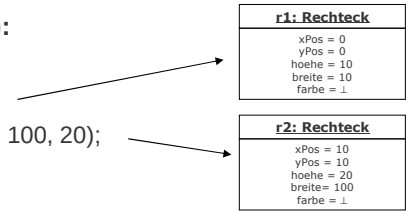
- Methodendefinitionen entsprechen Prozedur- bzw. Funktionsdefinitionen (s.o.); der Rückgabetyp **void** definiert Prozedur i. Ggs. zur Funktion)
- Die Sichtbarkeit der Attribute und Methoden einer Klasse wird mit den Modifkatoren *public (von außen zugänglicher Dienst)*, *protected* und *private (nur den Instanzen der Klasse selbst zugänglich)* festgelegt.
- Instanzen einer Klasse werden mit Hilfe des new-Operators (angewendet auf einen Konstruktor der Klasse) erzeugt.
- Konstruktoren sind spezielle Methoden zur Erzeugung von Instanzen einer Klasse. Versteht man Klassen als Objekte, so können Konstruktoren als sog. *Klassenmethoden* angesehen werden. In UML werden Klassenmethoden (und -attribute) durch Unterstreichung kenntlich gemacht bzw. von Instanzattributen/-methoden unterschieden.
- Die Bezeichner von Konstruktoren müssen dem Klassennamen entsprechen. Mehrere Konstruktoren mit unterschiedlichen Parameterlisten sind möglich.

Klassen und Objekte (9/17)

Klassen und Objekte in Java I (5/7):

Erzeugen eines Objekts (Beispiel):

- Rechteck r1 = **new** Rechteck();
- Rechteck r2 = **new** Rechteck (10, 10, 100, 20);



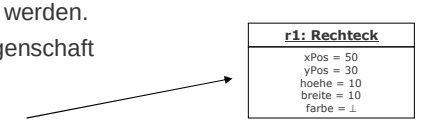
Zugriff auf Attribute eines Objekts:

Über die sogenannten *Referenzvariablen* kann auf die Attribute oder Methoden eines Objekts zugegriffen werden.

(Punkt-)Notation: objekt(variable).eigenschaft

Beispiele:

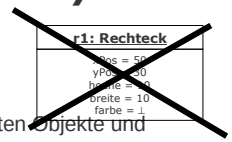
- r1.verschieben(50,30);
- r1.xPos = 30 // Direkter Verweis auf Attribut eines Objekts; nur unter besonderen Umständen erlaubt (public-Modifikator)
- **this**.xPos = 30 // this ist Schlüsselwort zur Kennzeichnung eines Zugriffs auf das Objekts selbst innerhalb einer Methode



Klassen und Objekte (10/17)

Klassen und Objekte in Java I (6/7):

Löschen eines Objekts:



- Objekte müssen *nicht explizit gelöscht* werden!
- Ein sog. *Garbage Collector* „sammelt" alle nicht mehr benötigten Objekte und gibt den durch sie belegten Speicher frei (vernichtet sie).
- Welche Objekte werden nicht mehr benötigt?

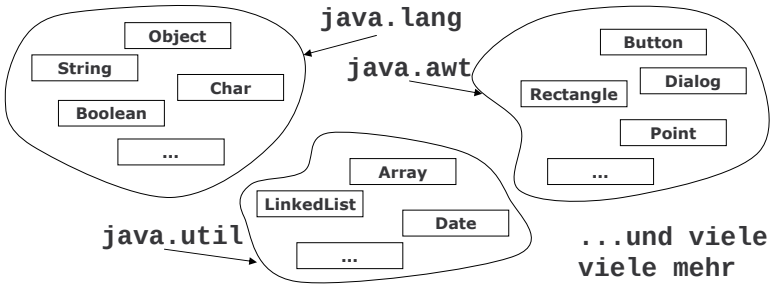
  Diejenigen, auf die *nicht mehr zugegriffen* werden kann!
  D.h. es gibt keine Globale Variable und kein dort gespeichertes Objekt, über das das betreffende Objekt erreichbar wäre (genauer: keine *transitive Hülle der Attribut-Referenzen* eines über eine Globale Variable zugänglichen Objekts enthält das betreffende Objekt)
- Um ein Objekt *der „Vernichtung" zuzuführen* müssen die Referenzen auf das Objekt gelöscht, d.h. mit „null" (Schlüsselwort) belegt werden.
- *Guter Programmierstil*: Pro Klasse eine *„finalize"-Methode*, die
  1. alle abhängigen Objekte löscht (ihnen finalize sendet)
  2. alle Attribute auf „null" setzt

Klassen und Objekte (11/17)

Klassen und Objekte in Java I (7/7):

Standardbibliotheken:

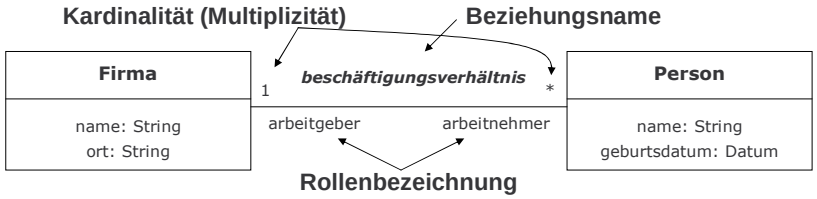
Ein wichtiger Aspekt der Wiederverwendbarkeit von Objekt- bzw. Klassenstrukturen bei *Objektorientierten Programmiersprachen* ist die Vorgabe mächtiger *Klassenbibliotheken*, d.h. Sammlungen nützlicher Klassen. In Java werden diese in sogenannten *Packages* verwaltet.



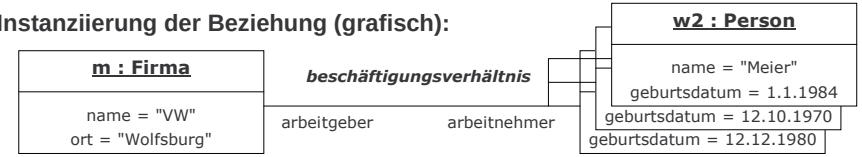
Klassen und Objekte (12/17)

Klassen und *Beziehungen (vereinfacht)*:

Regelmäßig zwischen Objekten auftretende Beziehungen können mit den Klassendefinitionen verknüpft werden:



Instanziierung der Beziehung (grafisch):



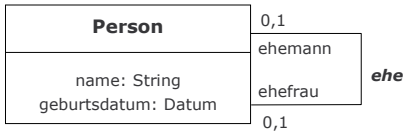
Klassen und Objekte (13/17)

Klassen und *Beziehungen (Fortsetzung)*:

Mögliche Kardinalitäten:

1	> Genau ein Beziehungspartner
0,1	> Kein oder maximal ein Beziehungspartner
0..4	> Zwischen null und vier Beziehungspartner
3,7	> genau drei oder genau 7 Beziehungspartner
0..*	> null oder beliebig viele Beziehungspartner (Gleichbedeutend: *)
*	> dito
1..*	> mindestens ein Beziehungspartner (ansonsten beliebig viele)

Beispiel:



Klassen und Objekte (14/17)

Klassen und Objekte in Java II: Beziehungen (1/4)

Direkte Umsetzung des Beziehungskonzeptes in Java nicht möglich! Java kennt „nur“ Objekte und Attribute!

Lösung:

Abbildung einer Beziehung in Attribute (nicht n,m-Beziehungen) und/oder spezielle Objekte (insb. n,m-Beziehungen):

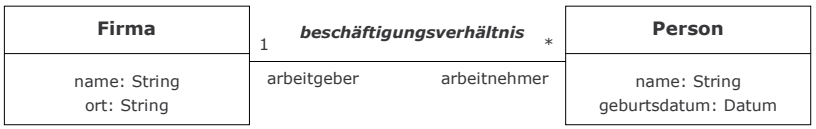
1 oder 0,1 > ein Attribut
> Name = Rollenname des Beziehungspartners (BP)
> Typ = Klasse des BP

n,m oder n..m mit $n \in IN$ und $m \in \{n, n+1,...\} \cup \{*\}$
> ein Attribut und ein Objekt zur Verwaltung des/r BP
> Attributname = Rollenname des BP
> Attribut-Typ = Array (bei fester Anzahl von BPs) oder beliebige Objektklasse zur Verwaltung von Mengen (ArrayList, LinkedList, Vector, eigene Klasse (Ehe) etc.)

Klassen und Objekte (15/17)

Klassen und Objekte in Java II: Beziehungen (2/4)

Modell:



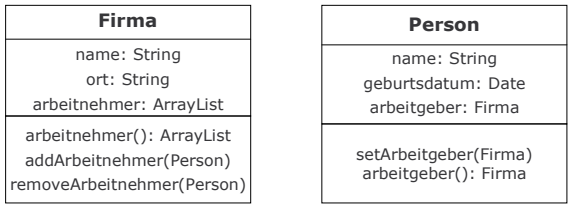
Implementierung in Java (Beispiel einer Variante):

Klassen und Objekte (16/17)

Klassen und Objekte in Java II: Beziehungen (3/4)

Guter Programmierstil:

Einrichten von *speziellen Diensten* zum Aufbau oder zum Abbau einer Beziehung zwischen zwei Objekten und zum Erfragen des/r BP in beiden an der Beziehung beteiligten Klassen.



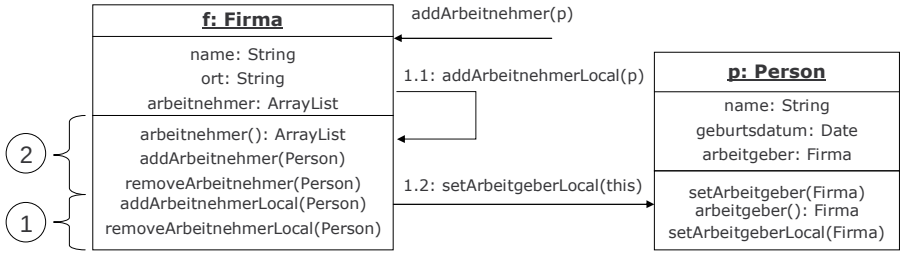
Klassen und Objekte (17/17)

Klassen und Objekte in Java II: Beziehungen (4/4)

Wichtig: Berücksichtigung *gegenseitiger* Beziehungen!

Lösung: Zwei Klassen von Manipulatoren *bei beiden* Beziehungspartnern

1. zum lokalen Eintragen des BP in das Rollenattribut des Objekts
2. zum zusätzlichen Eintrag des Objekts beim BP (der eigentliche Dienst)



Achtung: Gemischte Darstellung von Klasse und Objekt nicht UML-konform

Vererbung (1/15)

Bisher: Objekte können zu Klassen zusammengefasst werden, die Aussagen über Attribute, Verhalten und Beziehungen zu anderen Objekten enthalten (bzw. diese definieren).

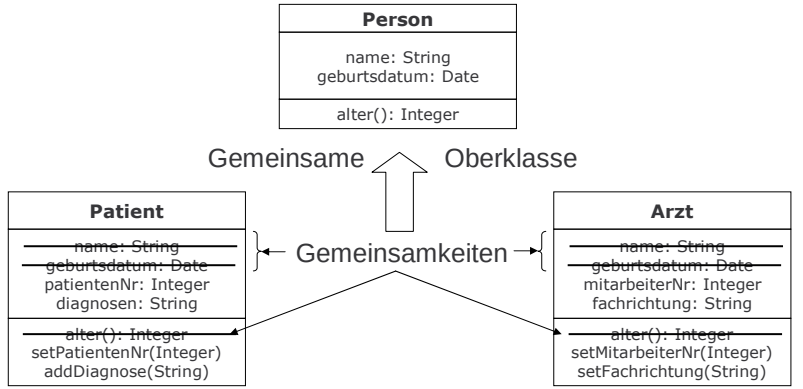
Beobachtung: Es gibt sehr viele Klassen, die bzgl. Attributen, Verhalten und Beziehungen *sehr ähnlich* sind (z.B. Person und Patient, Zahl und Rationale Zahl, Menge und Mehrfachmenge etc.)

Frage: Sollen für jede dieser Klasse diese gleichen bzw. ähnlichen Strukturen *immer wieder neu* angegeben/implementiert werden?

Antwort: NEIN!
Man versucht, zu ähnlichen Klassen eine gemeinsame Oberklasse zu finden,die die Ähnlichkeiten subsummiert (Attribute, Verhalten, Beziehungen) und ergänzt die Unterklassen nur um die Unterschiede (sie *erben* die Strukturen der Oberklasse).

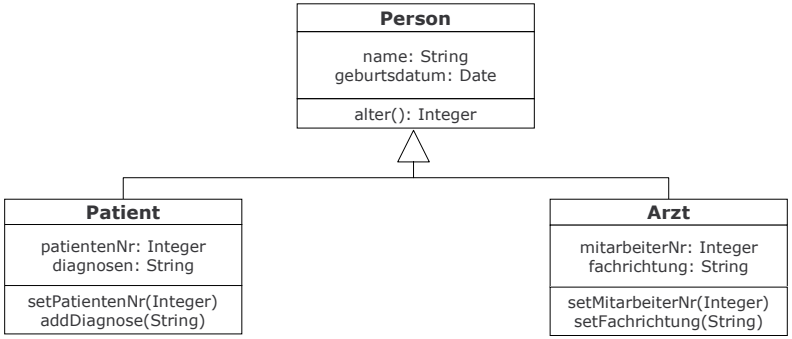
Vererbung (2/15)

Klassenbildung: „ähnliche Klassen“ werden zu einer Oberklasse zusammengefasst!



Vererbung (3/15)

Notation (UML)



Redeweise:

- Patient ist Unterklasse (Subclass) bzw. Spezialisierung von Person
- Person ist Oberklasse (Superclass) bzw. Generalisierung von Arzt und Patient

Vererbung (4/15)

Anmerkungen

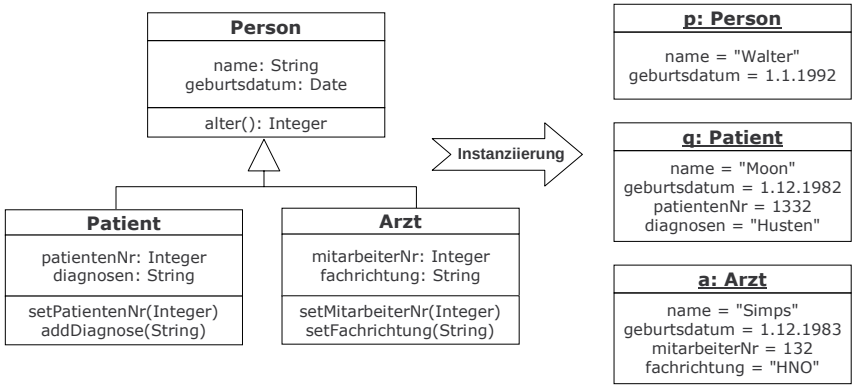
1. Eine Unterklasse *erbt* von ihrer Oberklasse (!rekursiv!) *alle Attribute und Methoden (Einschränkungen möglich: z.B. Java)* und kann diese um weitere Attribute und Methoden ergänzen.

   Erben heißt: die Attribute und Methoden der Oberklasse können in der Unterklasse verwendet werden, als wären sie in der Klasse selbst definiert.

2. Eine Klasse kann mehrere Oberklassen haben. In diesem Fall spricht man von *Mehrfachvererbung* (Multiple Inheritance) im Gegensatz zur *Einfachvererbung* (Singe Inheritance)
   Anmerkungen:
   - Problematisch ist die Behandlung von Definitionskonflikten (gleiche Attribute/Methoden in zwei Oberklassen).
   - Nicht alle OOPLs erlauben dies (z.B. Smalltalk nicht; Java nur eingeschränkt, C++ erlaubt Mehrfachvererbung).

Vererbung (5/15)

Beispiel: Zur gezeigten Vererbungshierarchie sind folgende Objekte möglich



Vererbung (6/15)

Auswertung von Methodenaufrufen zur Laufzeit I (1/2)

Das Senden einer Nachricht n an ein Objekt o löst zur Laufzeit eine Suche nach der passenden Methode aus:

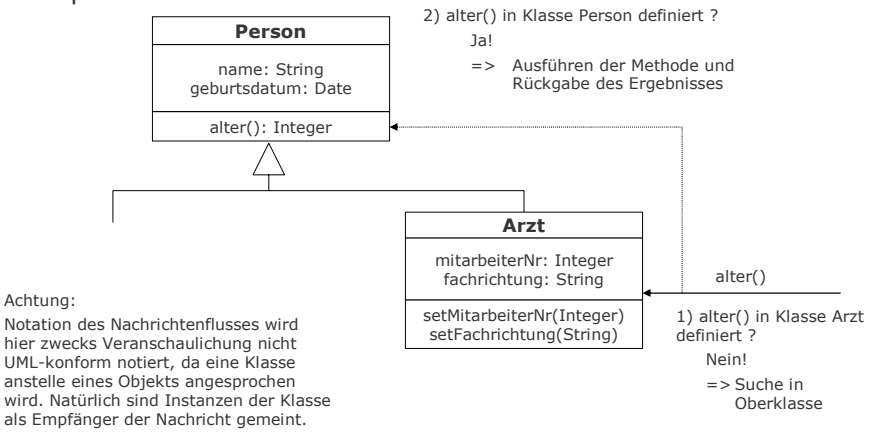
- Die Suche beginnt mit der Klassendefinition k des Objekts o.
- Wird in k keine zu n passende Methode gefunden, so wird rekursiv (d.h. entlang der Klassenhierarchie „aufwärts“ zur Wurzel) in der Oberklasse von k gesucht.
  "Passend" bezieht sich auf
  - Methodenname
  - Anzahl und Typ der Parameter
- Ist eine passende Methode gefunden, wird diese ausgeführt. Ansonsten kommt es zu einem Fehler.

Dies gilt für einfache Vererbung! Was gilt für multiple Vererbung?

Vererbung (7/15)

Auswertung von Methodenaufrufen zur Laufzeit I (2/2)

Beispiel:



Vererbung (8/15)

Polymorphie (1/3)

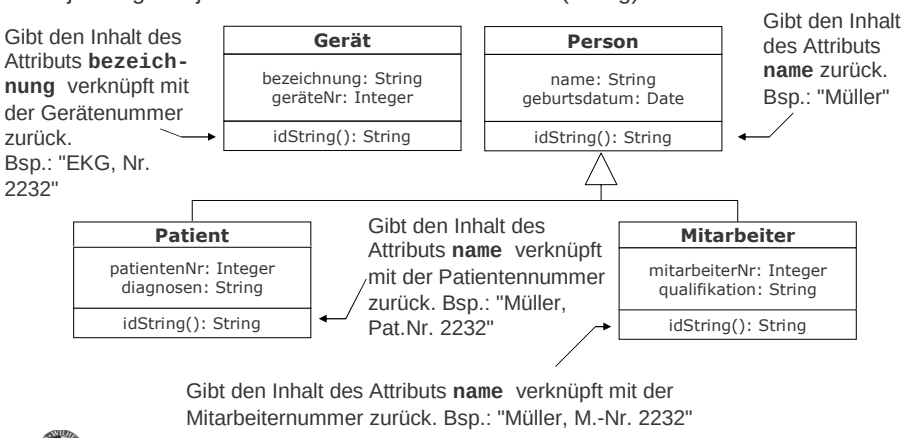
Methoden von Oberklassen können in Unterklassen *redefiniert* werden (Polymorphie)! Hierbei gibt es zwei Varianten:

1. **Überladen (Overloading)**
   Hierbei wird ein alternativer Programmblock unter dem *gleichen Methodennamen*, aber *mit unterschiedlicher Anzahl* und/oder *unterschiedlichem Typ der Parameter* angegeben. Allerdings bleibt der Ergebnistyp bei Funktionen gleich. Überladen ist *auch innerhalb* einer Klassendefinition möglich (s.z.B. Konstruktoren).
2. **Überschreiben (Overriding)**
   Hierbei wird unter dem gleichen Methodennamen ein alternativer Programmblock mit gleichem Parameterblock angegeben. Auch in bzgl. Vererbung „unabhängigen" Klassenbäumen sinnvoll!

Vererbung (9/15)

Polymorphie (2/3)

Beispiel (Überschreiben): Dienst **`idString()`** zur Rückgabe einer das jeweilige Objekt identifizierenden Zeichenkette (String)

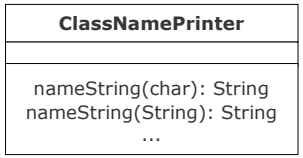


Vererbung (10/15)

Polymorphie (3/3)

Beispiel (Überladen; Java):

```
class ClassNamePrinter {

    public String nameString(char i) {

        return "Character"

    }

    public String
            nameString(String i) {

        return "String"

    }

    // etc.
}
```



Beispiel zur Auswertung:

```
...
ClassNamePrinter p
        = new ClassNamePrinter;
String result;
result = p.nameString('3')
            // result = "Character"
result = p.nameString("Character")
            // result = "String"
```

Vererbung (11/15)

Auswertung von Methodenaufrufen zur Laufzeit II
Oft ist es sinnvoll (bei Erweiterungen) in einer überschreibenden Methode auf die in einer Oberklasse definierten überschriebenen Methode zuzugreifen.

Hierzu wird i.d.R. ein spezielles Schlüsselwort verwendet: **super**

So erzwingt die Anweisung "super.idString()" in einer Methode der Klasse Mitarbeiter (s.o.), dass die Suche nach der zugehörigen Methode erst in der Oberklasse von Mitarbeiter beginnt (also in Klasse Person).

Beispiel (Klasse Mitarbeiter; Java, Implementierung von idString()):

```
...
public String idString () {
    return super.idString() + ", M.-Nr. " + mitarbeiterNr;
} ... // Gibt das im Beispiel (s.o.) angegebene Ergebnis aus
```


Vererbung (12/15)

Vererbung in Java

In Java wird Vererbung durch die folgende Notation ausgedrückt:

class *Unterklasse* **extends** *Oberklasse* {...}

Anmerkungen:

1. Wird keine Oberklasse angegeben, so ist automatisch die Klasse **Object** Oberklasse.
2. In Java gibt es *keine Multiple Vererbung*. Allenfalls kann eine Klasse (bzw. können Objekte einer Klasse) zusätzlich zur Spezialisierung einer Oberklasse verschiedene Schnittstellendefinitionen (Methodenspezifikationen) implementieren (>Programmierkurs).
3. Konstruktoren sind als Klassenmethoden zu verstehen. Sie werden nicht von Instanzen der Klasse ausgeführt!
4. Weitere Klassenmethoden und auch Klassenattribute können durch Verwendung des Modifikators "**static**" definiert werden!

Vererbung (13/15)

Abstrakte Klassen (1/3)

Problem:
Es soll eine Klasse definiert werden, deren Instanzen und Methoden in einem bestimmten Kontext verwendet werden sollen, deren Implementierung man aber nicht vorgeben bzw. bewusst offen halten möchte.

Beispiel:
Geometrische Objekte, die sich auf einer Darstellungsfläche mit Koordinatensystem selbst zeichnen können.

=> Name der Klasse und mind. die Deklaration der Dienste muss angegeben werden. Wird zu einem Dienst nur die Methodendeklaration angegeben, spricht man von einer *Abstrakten Methode* (Ggs.: *Konkrete Methode*)

=> Die Implementierung überlässt man Unterklassen dieser sogenannten *Abstrakten Klasse*.

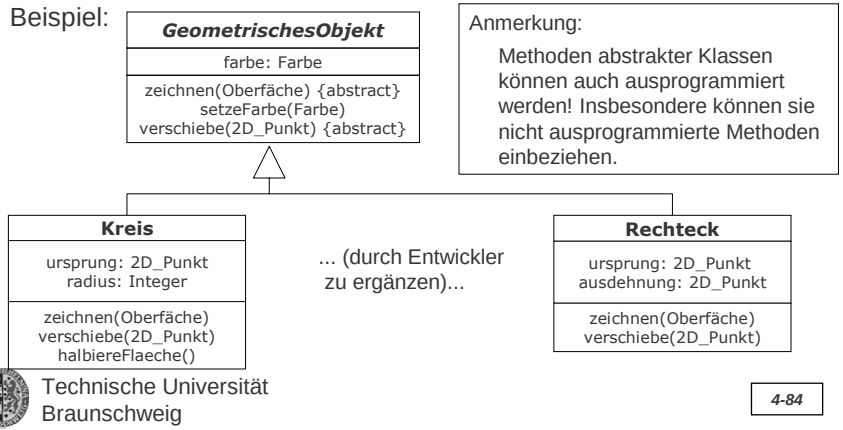
=> Instanzen einer Abstrakten Klasse selbst werden *nicht* erzeugt!

Vererbung (14/15)

Abstrakte Klassen (2/3)

Notation in UML:
Darstellung des Klassennamens in kursivem Schriftschnitt. Abstrakte Methoden werden mit dem Postfix {abstract} gekennzeichnet.

Beispiel:



Vererbung (15/15)

Abstrakte Klassen (3/3)

Abstrakte Klassen in Java:
Ergänzung des Schlüsselworts abstract bei der Klassendefinition bzw. bei der Methodendeklaration. Bei Abstrakten Methoden darf kein Anweisungsblock folgen.

Beispiel:

```
abstract class GeometrischesObjekt {
    Color farbe;
    public abstract void zeichnen (Graphics screen);
    public void setzeFarbe (Color f) {
        farbe = f;
    }
    // etc.
}
```


Umsetzung von ADTs (1/5)

Idee: ADT ⇨ Klasse

Vorgehen:

type *Name des ADT* → **Klasse**

import *Liste von ADT* → attribut (Beziehung): Typ

operators

Liste von Operatoren
$f : s_1 \ldots s_n \rightarrow s$:

- *Konstruktoren*
- *Selektoren*
- *Manipulatoren*

→ operation (PK_1,..., PK_n): EK
$f : s_1 \ldots s_n \rightarrow s$:

- *Konstruktoren* → Instanzerzeugende (Klassen-)Methoden (z.B. Konstruktoren in Java)
- *Selektoren* → Auswertende Funktionen mit Lesezugriff auf Attribute/Beziehungen
- *Manipulatoren* → Verändernder Zugriff auf Attribute/Beziehungen

axioms

Liste von Axiomen zu den Operatoren → Oft keine explizite Entsprechung (Java > Programmcode, UML>Kardinalitäten, OCL ...)

Umsetzung von ADTs (2/5)

Beispiel: ADT Rat(ionale Zahl) ⇨ Klasse RationaleZahl

type Rat

import Int, Bool

operators //Auswahl:

mk_rat: Int × Int → Rat
nominator: Rat → Int
denominator: Rat → Int
equal: Rat × Rat → Bool
add: Rat × Rat → Rat
normalize: Rat → Rat

Konstruktor

RationaleZahl
nom: Integer denom: Integer
RationaleZahl(Integer, Integer) nominator(): Integer denominator(): Integer equal(RationaleZahl): Boolean add(RationaleZahl): RationaleZahl normalize()

axioms

∅

Umsetzung von ADTs (3/5)

Beispiel (Fortsetzung): Implementierung der Klasse RationaleZahl in Java (1/3)

RationaleZahl
nom: Integer denom: Integer
RationaleZahl(Integer, Integer) (de-)nominator(): Integer equal(RationaleZahl): Boolean add(RationaleZahl): RationaleZahl normalize()

```
public class RationaleZahl {
//Attributdeklarationen
   int nom, denom;
//Konstruktoren
   public RationaleZahl(int n, int d) {
         nom = n; denom = d;
         normalize();
   }
//Selektoren; (de-)nominator() verkürzt
   public int nom() {
         return nom;
   }     //...
```

```
// Fortsetzung Selektoren
   public int denom() {
      return denom;
   }
   public boolean equals
         (RationaleZahl r) {
      return this.nom() * r.denom() ==
         this.denom() * r.nom();
   } //...
```

Umsetzung von ADTs (4/5)

Beispiel (Fortsetzung): Implementierung der Klasse RationaleZahl in Java (2/3)

RationaleZahl
nom: Integer denom: Integer
...
add(RationaleZahl): RationaleZahl normalize()

```
// Manipulatoren
   public RationaleZahl
          add(RationaleZahl r) {
      int n,d;
      n = this.nom() * r.denom()  +
           r.nom() * this.denom();
      d = this.denom() * r.denom();
      return new RationaleZahl(n,d);
   }
   puplic void normalize() {
      int g = ggt(nom, denom);
      if (denom > 0) {
          nom = nom / g;
          denom = denom / g; }
          else if (denom < 0) {
          nom = -nom / g;
          denom = -denom / g; }     }
// Private Hilfsfunktion ggt
    private int ggt(int z1, int z2) {
       int r = 1;
       while (r != 0) {
          r = z1 % z2;     // % = mod
          z1 = z2; z2 = r; }
       return z1;   }
}
```


Umsetzung von ADTs (5/5)

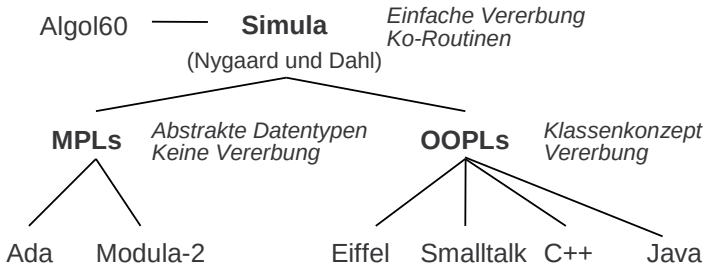
Beispiel (Fortsetzung):
Implementierung der Klasse RationaleZahl in Java (3/3)

Anmerkungen:

1. Die Implementierung ist aus didaktischen Gründen sehr einfach gehalten und verzichtet u.a. auf
   a. Ausnahme- bzw. Fehlerbehandlung (z.B. Zähler = 0) und
   b. Konvertierungen (z.B. im Zusammenhang mit dem Test auf Gleichheit: hier könnte auch mit Integer-Werten etc. verglichen werden; dies würde Überschreibungen der Equals-Methode nötig machen: public boolean equals(Integer)...).
2. Der Zugriff auf Attribute eines Objekts sollte der Wartbarkeit wegen immer über entsprechende Methoden (insb. set... und get...) erfolgen (i. Ggs. zur Impl. oben; dort ist zwecks Übereinstimmung mit ADT bzw. der Kürze der Darstellung halber anders vorgegangen worden). Hier ist zwischen „Sauberkeit/Stabilität“ der Implementierung und ihrer Effizienz abzuwägen.

Objektorientierte Programmiersprachen

Historie:



... sowie verschiedene OO-Derivate anderer Programmiersprachen (Basic, Pascal etc.)

Anmerkung: *Objektorientierte Datenbanken (OODB)* erlauben die dauerhafte Speicherung von Objekten und ggf. die Ausführung von Diensten durch die DB

4.6 Grundlegende Datenstrukturen

Überführung der z.T. bereits behandelten ADT in Objekt- bzw. Klassenstrukturen:

1. Modellierung (UML)
2. Implementierung (Java; auszugsweise)
   Achtung: für einige der behandelten ADT existieren vorgegebene Java-Klassen in entspr. Packages, die von der besprochenen Implementierung abweichen.

Behandelt werden:

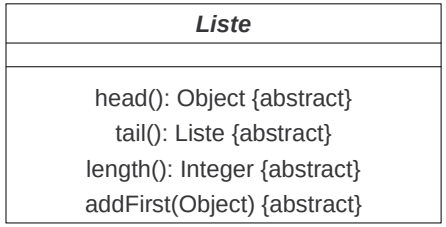
1. Listen
2. Keller bzw. Stapel (Stack)
3. Warteschlangen (Queue)
4. Mengen (Set)
5. Binärbäume (Tree)

Listen (1/11)

type List(T)
import Nat
operators
[ ]: → List
_:_ : T×List → List
head : List → T
tail : List → List
length : List → Nat
axioms ∀ *l*: List, ∀ *x*: T
head (*x*:*l*) = *x*
tail (*x*:*l*) = *l*
length([])=0
length(*x*:*l*)
= succ(length(*l*))

Definition einer abstrakten Klasse Liste:

(Gibt die Schnittstelle vor und kann durch ggf. verschiedene Implementierungen realisiert werden)



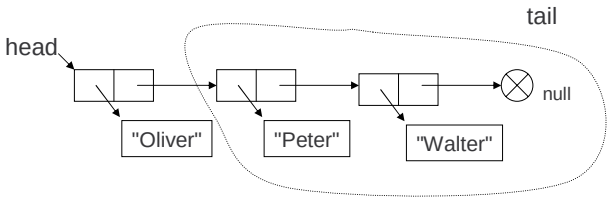
Listen (2/11)

Implementierung (1/8)

1. Idee: als Verkettung einzelner Objekte (Einfach verkettete Liste)

Beispiel: Liste von Namen

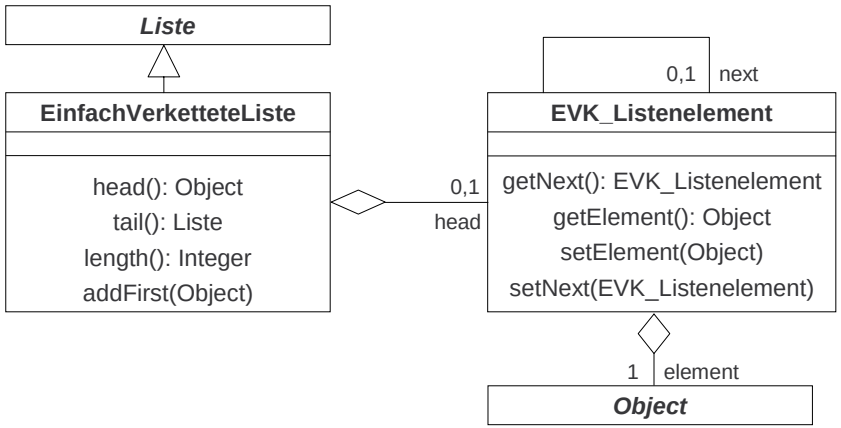
z.B. ["Oliver", "Peter", "Walter"]




Knoten (engl. Node; hier: Listenelement)

Listen (3/11)

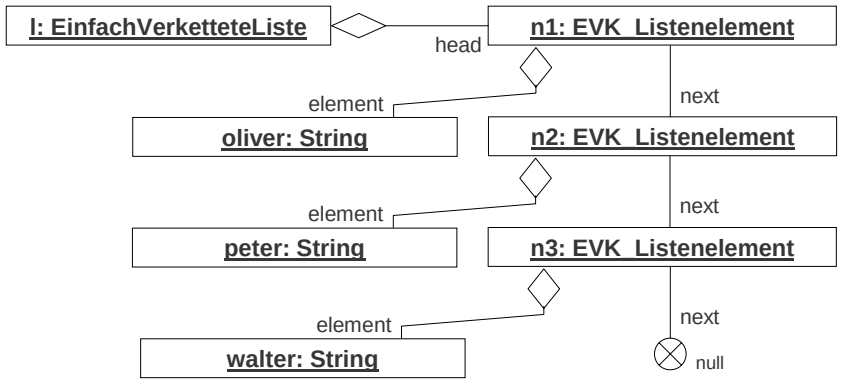
Implementierung (2/8): Einfach verkettete Liste (Modell)



Listen (4/11)

Implementierung (3/8): Einfach verkettete Liste (Modell)

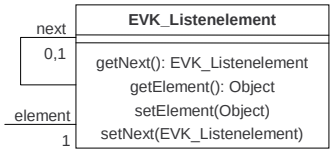
Beispielinstanziierung:



Listen (5/11)

Implementierung (4/8): Einfach verkettete Liste (Java)

```
public class EVK_Listenelement
{
//Attributdeklarationen
    protected Object element;
    protected EVK_Listenelement next;
//Konstruktoren
    public EVK_Listenelement() {
        element = null; next = null;
    }
    public EVK_Listenelement
      (Object e, EVK_Listenelement n) {
        element = e; next = n;
    } //...
```

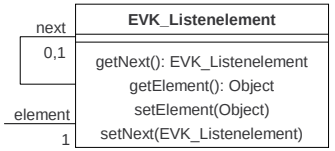



```
// Selektoren
    public Object getElement() {
        return element;
    }
    public EVK_Listenelement
          getNext() {
        return next;
    }       //...
```


Listen (6/11)

Implementierung (5/8): Einfach verkettete Liste (Java)

```
//..Fortsetzung
// Manipulatoren
  public void setElement(Object o) {
        element = o;
  }
  public void setNext(EVK_Listenelement n)
  {     next = n;
  }
// der Vollständigkeit (Sauberkeit) halber:
  protected void finalize()
        throws Throwable {
        next = null; element = null;
        super.finalize() }
}
```



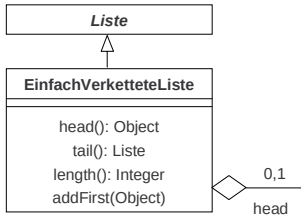
Anmerkungen zu finalize:

- Das Schlüsselwort **throws** kennzeichnet die Fehler- bzw. Ausnahme-behandlung in Java (wird hier nicht weiter betrachtet)
- **protected** bedeutet i.W.: in der Klasse selbst und in Unterklassen verwendbar

Listen (7/11)

Implementierung (6/8): Einfach verkettete Liste (Java)

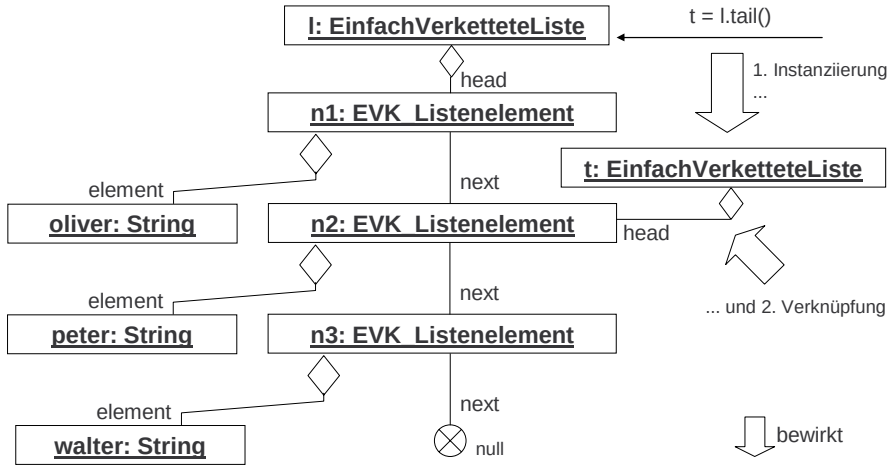
```
public class EinfachVerketteteListe
    extends Liste
{
//Attributdeklarationen
    protected EVK_Listenelement head;
//Konstruktoren
    public EinfachVerketteteListe() {
        head = null; // Standard
    }
// Selektoren
    private boolean isEmpty() {
        return head == null;
    } //...
```




```
// Fortsetzung Selektoren
    public Object head() {
        if (! this.isEmpty())
            return head.getElement();
        else
            return null;
    }
```


Listen (8/11)

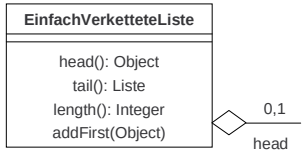
Einschub: Modell zur Implementierung der tail()-Methode



Listen (9/11)

Implementierung (7/8): Einfach verkettete Liste (Java)

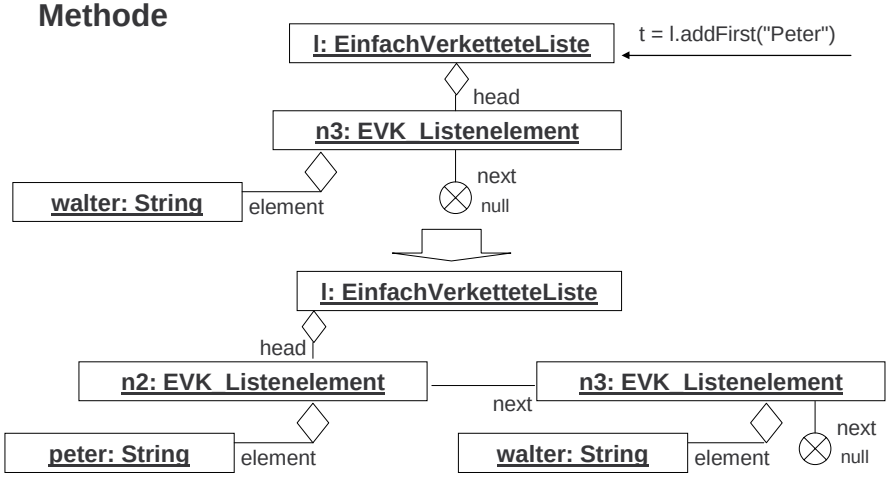
```
// Fortsetzung Selektoren
    public Liste tail() {
    // Erzeugt neue Liste mit Zeiger auf
    // das zweite Element der ursprüng-
    // lichen Liste
        EinfachVerketteteListe evk
          = new EinfachVerketteteListe();
        if (! this.isEmpty())
            evk.head = head.getNext();
        return evk;
     // Achtung: Ursprüngliche Liste samt
    // Kopfelement bleibt erhalten
    }   // ...
```




```
// Fortsetzung Selektoren
    public int length() {
        int z = 0;
        EVK_Listenelement n = head;
        while (n != null) {
                n=n.getNext();
                z = z + 1;
        }
        return z;
    } //...
```


Listen (10/11)

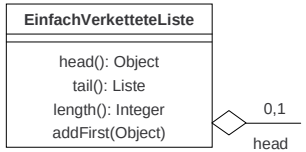
Einschub: Modell zur Implementierung der addFirst()-Methode



Listen (11/11)

Implementierung (8/8): Einfach verkettete Liste (Java)

```
// Manipulatoren
    public void addFirst(Object o) {
    // Erzeugt neues Listenelement mit
    // Zeiger auf das Objekt und fügt es
    // als Vorgänger des ursprünglichen
    // Listenkopfs ein
        EVK_Listenelement n =
            new EVK_Listenelement();
        n.setElement(o);
        n.setNext(head);
        head = n;
    }
}
```



Achtung! Dies ist eine unvollständige Implementierung:

- tail() ist die einzige Methode zur Reduzierung einer Liste. Besser wäre eine Methode removeFirst() o.ä.
- Weitere nützliche Funktionen wären sinnvoll/nötig (s.u.).

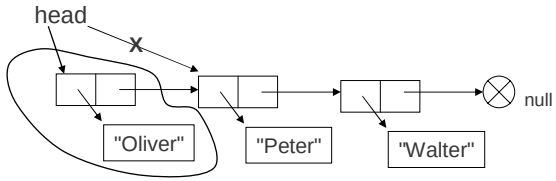
Doppelt verkettete Listen

Analyse ausgewählter Listenoperationen bei EVK (1/3)

1. **addFirst(Object)**
   //Anfügen eines Objekts am Listenkopf
   Ablauf:
   a. Erzeugen eines neuen Knoten: O(1)*
   b. Neuen Knoten auf ehemaligen head verweisen lassen: O(1)
   c. "Umlenken" von head: O(1)

   Insgesamt: O(1)

```
public void addFirst(Object o) {
    EVK_Listenelement n =
            new EVK_Listenelement();
    n.setElement(o);
    n.setNext(head);
    head = n;
}
```

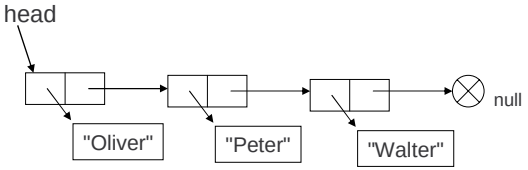


*Wir betrachten im folgenden den average Case

Analyse ausgewählter Listenoperationen bei EVK (2/3)

2. head(): Object
//Rückgabe des Objekts am Listenkopf
Ablauf:
a. Fallunterscheidung: O(1)
b. Rückgabe von null: O(1)
c. Objekt des ersten Listenelements zurückgeben: O(1)
Insgesamt: O(1)

```
public Object head() {
    if (! this.isEmpty())
        return head.getElement();
    else
        return null;
}
```



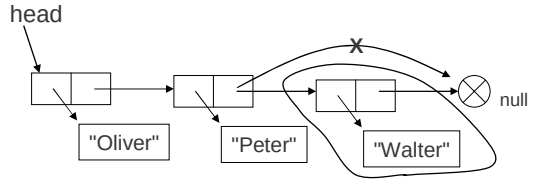
Analyse ausgewählter Listenoperationen bei EVK (3/3)

3. addLast()
// Anfügen eines Objekts am
// Listenende
Ablauf:

a. "Navigieren" zum letzten Knoten: O(n)*
b. Erzeugen des neuen Knotens: O(1)
c. "Einhängen" des neuen Knotens: O(1)
Insgesamt: O(n)

```
public void addLast(Object o) {
    if (this.isEmpty()) this.addFirst(o);
    else {
        EVK_Listenelement n, ok;
        n = head;
        while (n.getNext() != null)
            n = n.getNext();
        ok = new EVK_Listenelement();
        ok.setElement(o);
        n.setNext(ok);
    }
```

*n = Länge der Liste (auch im Folgenden)

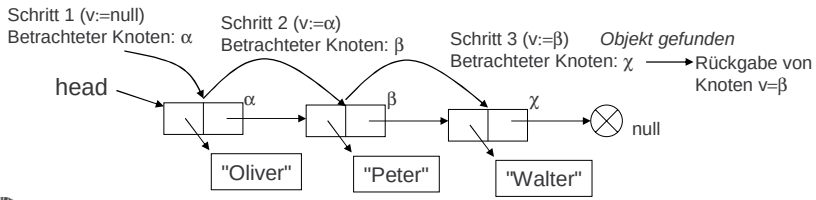


Finden von Vorgängern eines Knotens in EVK (1/3)

1. Finden des direkten Vorgängers eines Knoten k in der Liste
Algorithmus (ohne Ausnahmebehandlung):
a. "Navigieren" zum Knoten k (dabei Zeiger v auf Vorgängerknoten des aktuell betrachteten Knotens mitführen): O(n)*
b. Rückgabe des Knoten v: O(1)
Insgesamt: O(n)

*Ohne (den trivialen) Beweis.

Bsp. Bestimmung des Vorgängers von χ ("Walter "): β ("Peter"):

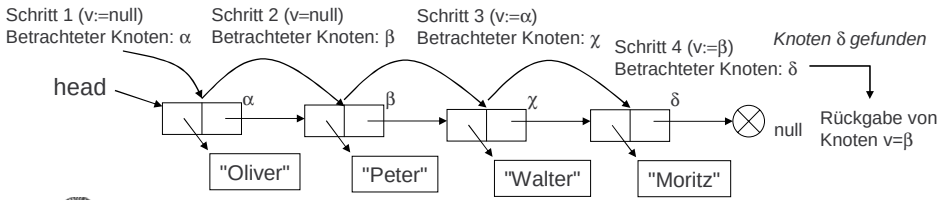


Finden von Vorgängern eines Knotens in EVK (2/3)

2. Finden des m'ten Vorgängers eines Knoten k in der Liste

1. Idee: Änderung des Algorithmus
Zeiger v wird erst *nach* (m+1)-Schritten (d.h. nach m-fachem „Weiternavigieren" des Zeigers auf das betrachtete Element) mit dem ersten Knoten der Liste verknüpft und erst danach ebenfalls mit weitergesetzt (Laufzeit: O(n)).

Bsp. Bestimmung des 2'ten Vorgängers von δ ("Moritz"): β ("Peter"):

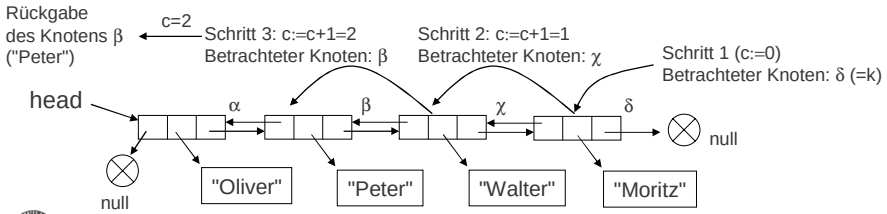


Finden von Vorgängern eines Knotens in EVK (3/3)

2. **Idee: Änderung der Datenstruktur (Doppelt verkettete Liste)**
   Jeder Knoten wird um einen Zeiger auf das Vorgängerelement der Liste ergänzt.

   Die Suche nach dem m'ten Vorgänger von Knoten k erfolgt dann ausgehend vom Knoten k unter Verwendung eines Zählers c
   (Laufzeit: O(m) d.h. unabhängig von der Länge der Liste!).

Bsp. Bestimmung des 2'ten Vorgängers von δ ("Moritz"): β ("Peter"):



Doppelt verkettete Listen (DVK)

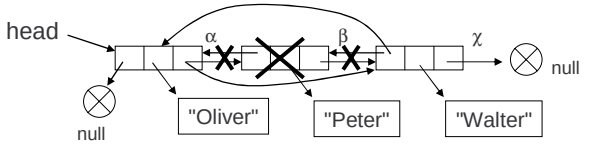
Vorteil:

- Navigation in der Liste bzw. Zugriff auf in der Liste vor einem Knoten liegende Knoten wird vereinfacht.

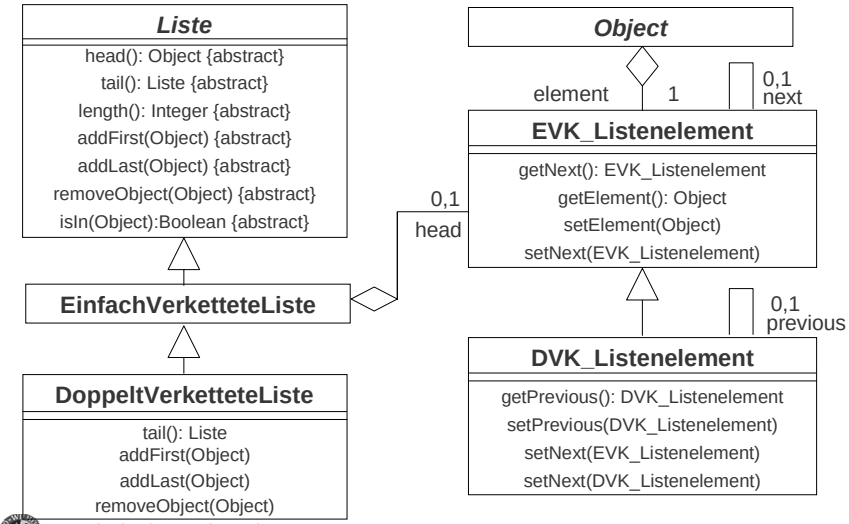
Nachteil:

- Verwaltung des zweiten Zeigers bei Einfüge- und Löschoperationen ist (etwas) aufwändiger.

Bsp. Löschen des Knotens β

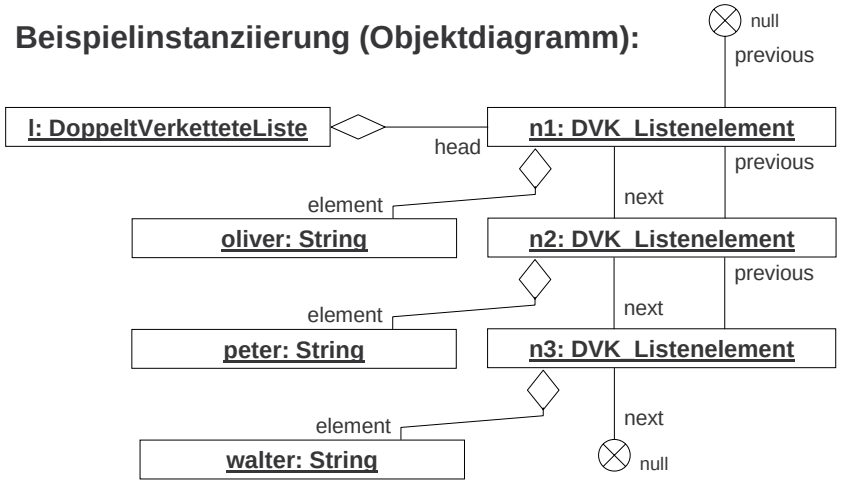


DVK (OO-Umsetzung 1/4)



DVK (OO-Umsetzung 2/4)

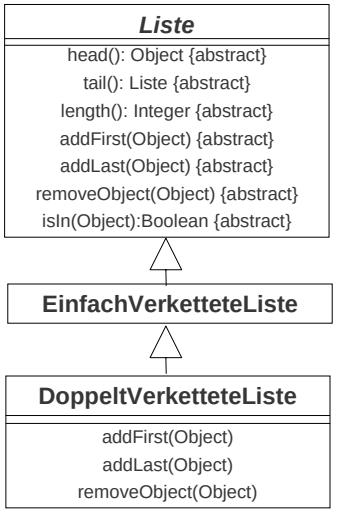
Beispielinstanziierung (Objektdiagramm):



DVK (OO-Umsetzung 3/4)

Anmerkungen zur Klasse DoppeltVerketteteListe

- Definition der Klasse als Unterklasse von EinfachVerketteteListe bietet sich an, da viele Eigenschaften ähnlich sind und übernommen werden können.
- Insbesondere werden die Dienste *head(): Object, length(): Integer* und *isIn(Object):Boolean* der Klasse EinfachVerketteteListe ohne weiteres funktionieren.
- *tail(): Liste, addFirst(Object), addLast(Object)* und *removeObject(Object)* müssen *überschrieben* werden, da Knotenerzeugung und -manipulation auf die Knotenklasse DVK_Listenelement abzustimmen sind.



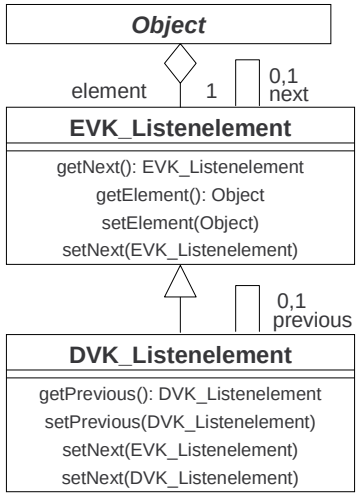
DVK (OO-Umsetzung 4/4)

Anmerkungen zur Klasse DVK_Listenelement

- als *next*-Knoten sind ausschließlich DVK_Listenelemente zulässig.

  Um dies sicherzustellen kann der Dienst *setNext(EVK_Listenelement)* dahingehend *überschrieben* werden, dass er einen Fehler „erzeugt".

  Des Weiteren wird er mit *setNext(DVK_Listenelement)* -und korrekter Funktion- *überladen*.
- Bei *getNext()* sind wg. des Rückgabetyps EVK_Listenelement ggf. Casting-Operationen nötig, um auf die Eigenschaften des DVK_Listenelements zugreifen zu können.

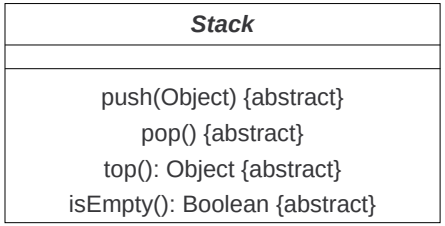


Stack (1/6)

type Stack(T)
import Bool
operators
empty: → Stack
push: Stack×T → Stack
pop : Stack → Stack
top : Stack → T
empty?: Stack → Bool
axioms ∀ *s*: Stack, ∀ *x*: T
pop(push(*s*,*x*)) = *s*
top(push(*s*,*x*)) = *x*
empty?(empty)= true
empty?(push(*s*,*x*)) = false

Definition einer abstrakten Klasse Stack:

(Gibt die Schnittstelle vor und kann durch ggf. verschiedene Implementierungen realisiert werden)

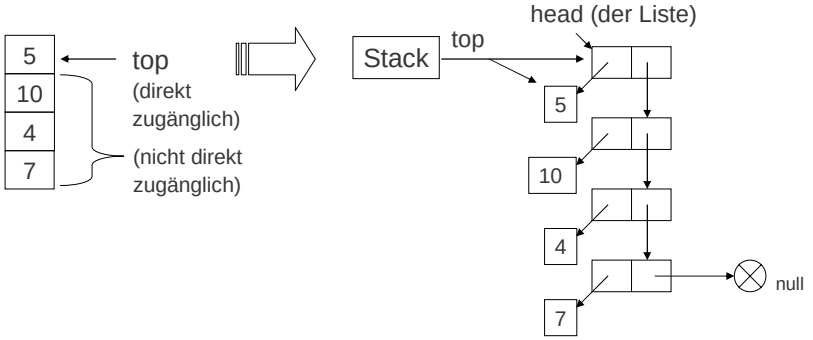


Stack (2/6)

Implementierung (1/5)

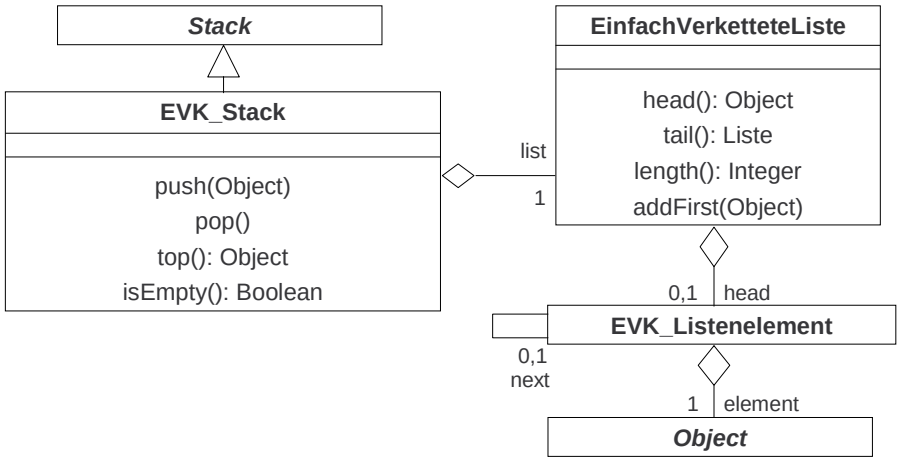
1. Idee: wie bei ADT Stack mit (einfach verketteter) Liste

Beispiel: Stack von Zahlen



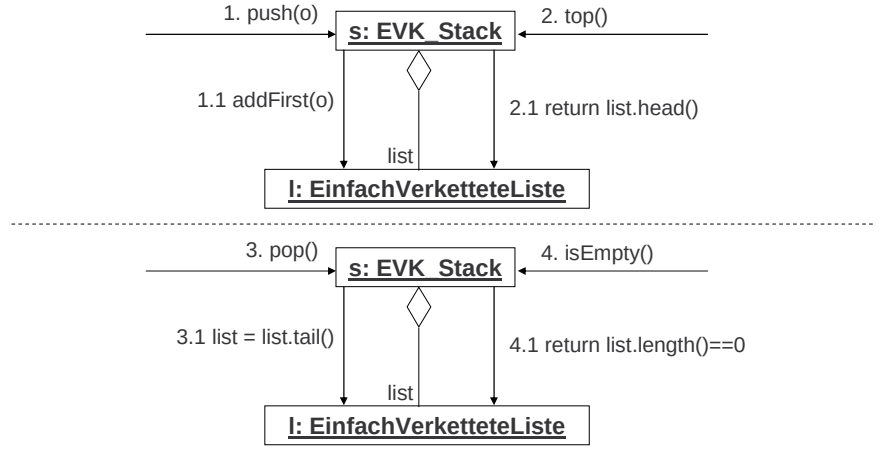
Stack (3/6)

Implementierung (2/5): Klassenmodell



Stack (4/6)

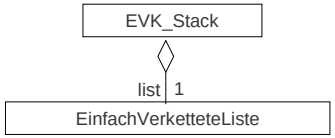
Implementierung (3/5): Kollaborationsdiagramme



Stack (5/6)

Implementierung (4/5): Java-Implementierung

```
public class EVK_Stack extends
    Stack
{
//Attributdeklarationen
    protected EinfachVerketteteListe list;
//Konstruktoren
    public EVK_Stack() {
        list = new EinfachVerketteteListe();
    }
// Selektoren
    protected EinfachVerketteteListe
            getList()  {
        return list;
    } //...
```




```
// Selektoren Fortsetzung
    public boolean isEmpty() {
        return this.getList().length()==0;
    }
    public Object top() {
        return
            this.getList().head();
    }
//...
```


Stack (6/6)

Implementierung (5/5): Java-Implementierung

```
// Manipulatoren
    public void push(Object o) {
        this.getList().addFirst(o);
    }
    public void pop () {
        list = (EinfachVerketteteListe) this.getList().tail();
    }
}
```

Sogenannter Cast-Operator zur Typwandlung (wird benötigt, da für tail() Liste als Rückgabetyp, für das Attribut list als Typ aber EinfachVerketteteListe definiert ist).

Anmerkungen:

- Folgen von Nachrichten werden in Java von links nach rechts ausgewertet. D.h. $o.n_1().n_2().....n_n()$ entspricht $(...(o.n_1).n_2)...n_n())$. Eine explizite Klammerung ist nicht nötig.
- Der Aufruf einer Methode bindet stärker als unäre oder binäre Operatoren. z.B. 2+4.sqrt() ergibt 4 (wenn sqrt() = Methode zur Berechnung der Wurzel)

Queue (1/8)

type Queue(T)
import Bool
operators
empty: → Queue
enter: Queue ×T → Queue
leave: Queue → Queue
front: Queue → T
empty?: Queue → Bool
axioms ∀ *q*: Queue, ∀ *x*: T
leave(enter(empty,*x*)) = empty
leave(enter(enter(*q*,*x*), *y*))
= enter(leave(enter(*q*,*x*)), *y*)
front(enter(empty,*x*)) = *x*
front(enter(enter(*q*,*x*),*y*))
= front(enter(*q*,*x*))
empty?(empty) = true
empty? (enter(*q*,*x*)) = false

Definition einer abstrakten Klasse Queue:

(Gibt die Schnittstelle vor und kann durch ggf. verschiedene Implementierungen realisiert werden)

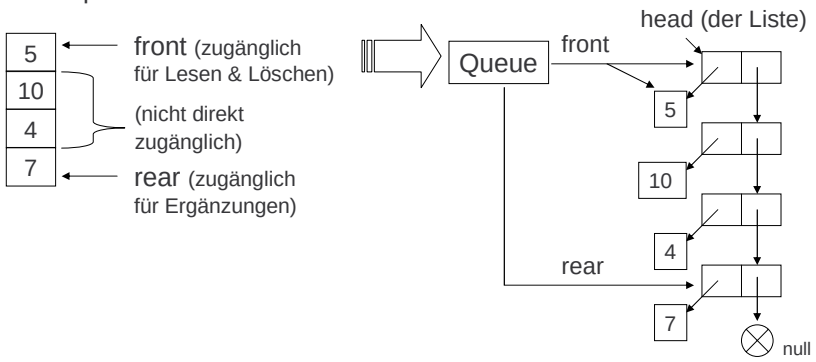
Queue
enter(Object) {abstract}
leave() {abstract}
front(): Object {abstract}
isEmpty(): Boolean {abstract}

Queue (2/8)

Implementierung (1/6)

1. Idee: ähnlich zu ADT Stack mit (einfach verketteter) Liste

Beispiel: Stack von Zahlen



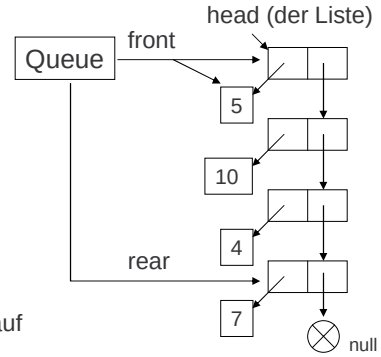
Queue (3/8)

Implementierung (2/6)

Problem: Anfügen neuer Elemente am Ende der Liste benötigt.

Erste Idee: *Das Queue-Objekt*

1. durchsucht die Liste, d.h. durchwandert alle Listenelemente bis zum letzten,
2. erzeugt ein neues Listenelement mit dem einzufügenden Objekt
3. lässt das vormals letzte Listenelement auf das nun neue Listenelement verweisen.

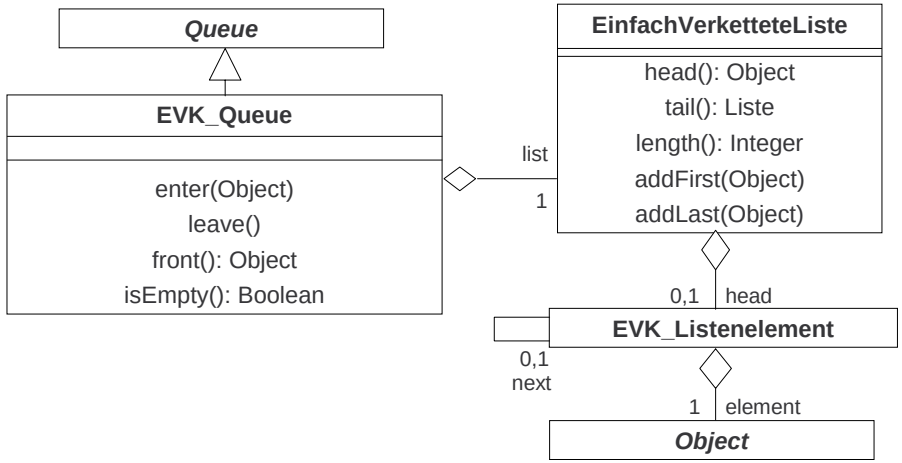


Diese Idee ist schlecht, da sie gegen das Prinzip der *Kapselung* verstößt. Es sollte der Liste überlassen sein, ein Objekt an ihrem Ende anzufügen.

=> Bessere Lösung: Ergänzen der Klasse *EinfachVerketteteListe* um einen addLast()-Dienst, der genau dieses leistet.

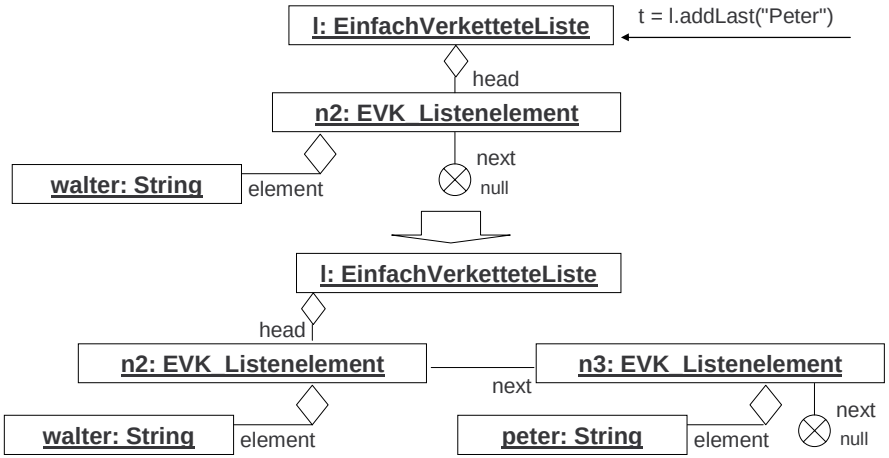
Queue (4/8)

Implementierung (3/6): Klassenmodell



Queue (5/8)

Implementierung (4/6): addLast(Object) im Modell



Queue (6/8)

Implementierung (5/6): Ergänzung der einfach verketteten Liste um den Dienst addLast(Object) (Java)

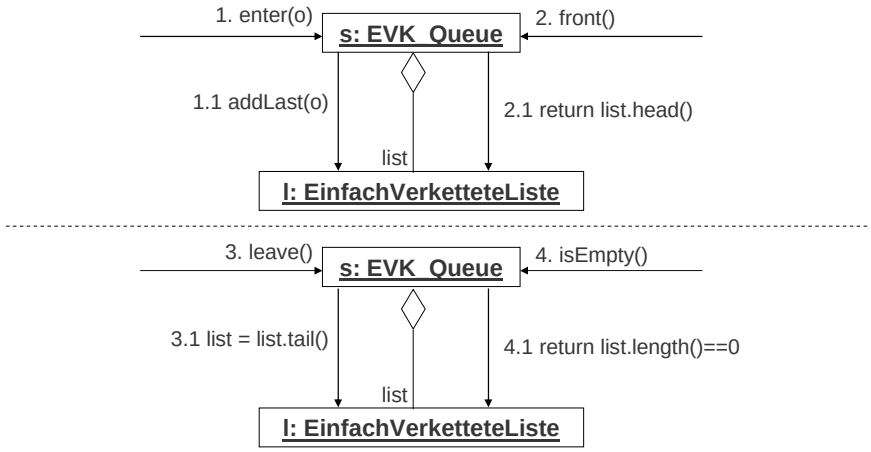
```
// Ergänzung Klassendefinition
// EinfachVerketteteListe
// Fortsetzung Manipulatoren
    public void addLast(Object o) {
        if (this.isEmpty()) this.addFirst(o);
        else {
            EVK_Listenelement n, ok;
            n = head;
            while (n.getNext() != null)
                n = n.getNext();
            ok = new EVK_Listenelement();
            ok.setElement(o);
            n.setNext(ok);
        }
    } //...
```

Kommentare:

- Erste Fallunterscheidung testet auf leere Liste. In diesem Fall kann das Objekt am Anfang eingefügt werden.
- Andernfalls wird die Liste durchlaufen und ein neues Element an das ursprünglich letzte Element „gehängt“.

Queue (7/8)

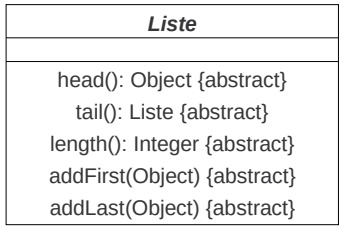
Implementierung (6/6): Kollaborationsdiagramme



Queue (8/8)

Anmerkungen:

- Java-Implementierung analog zu Stack.
- Andere (bessere) Implementierungen sind möglich.
- Anstelle der Ergänzung der Klasse EinfachVerketteteListe hätte auch eine eigene Unterklasse definiert werden können. Letzteres ist das normale Vorgehen bei Ergänzungen von Klassen, die nicht im eigenen Verantwortungsbereich liegen (z.B. Klassen aus Standardbibliotheken).
- Da der Dienst addLast() zum Einfügen eines Elements an das Ende einer Liste generell bei Listen sinnvoll ist, sollte er auch als abstrakte Methode in der Klasse Liste definiert sein:



Set (1/8)

type Set(T)
import Bool
operators
∅ : → Set
empty? : Set → Bool
insert : Set ×T→ Set
delete : Set ×T→ Set
is_in : Set ×T→ Bool
axioms ∀ *s*: Set, ∀ *x,y*: T
(s.o.)

Definition einer abstrakten Klasse Set (Menge):

(Gibt die Schnittstelle vor und kann durch ggf. verschiedene Implementierungen realisiert werden)

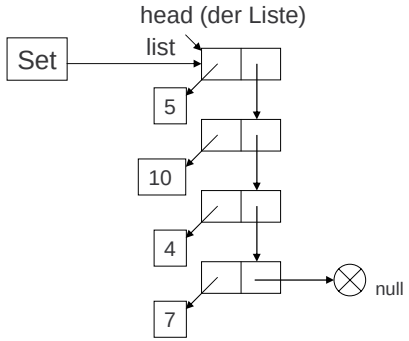
Set
insert(Object) {abstract}
delete(Object) {abstract}
isIn(Object): Boolean {abstract}
isEmpty(): Boolean {abstract}

Set (2/8)

Implementierung (1/5)

1. Idee: wie bei ADT Set mit (einfach verketteter) Liste

Beispiel: Menge von Zahlen {5, 10, 4, 7}



Problem 1: Entfernen von Elementen aus der Liste

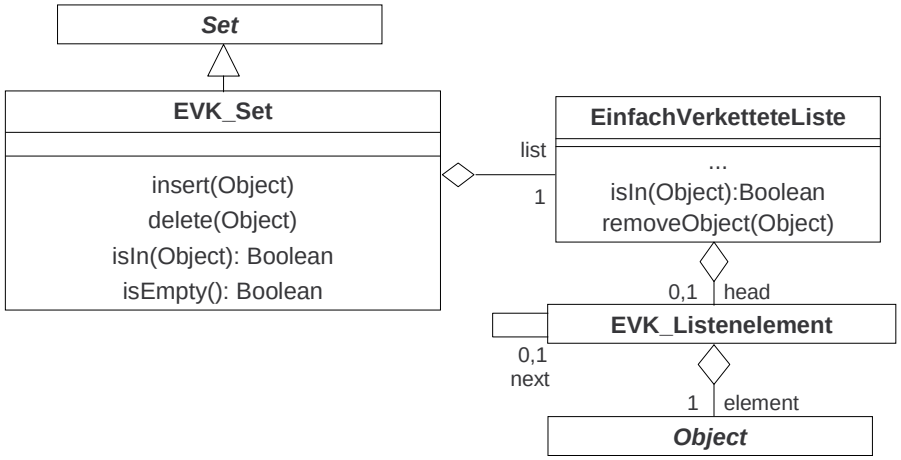
(Eine) Lösung: Ergänzen der Klasse *EinfachVerketteteListe* um einen removeObject()-Dienst

Problem 2: Finden eines Elements in der Liste

(Eine) Lösung: Ergänzen der Klasse *EinfachVerketteteListe* um einen isIn()-Dienst

Set (3/8)

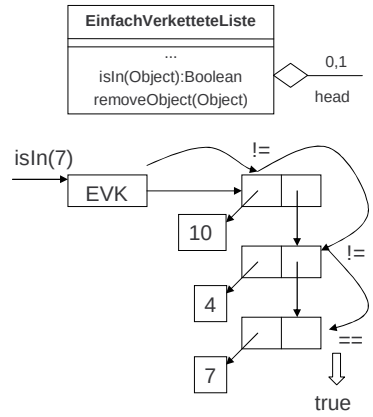
Implementierung (2/5): Klassenmodell



Set (4/8)

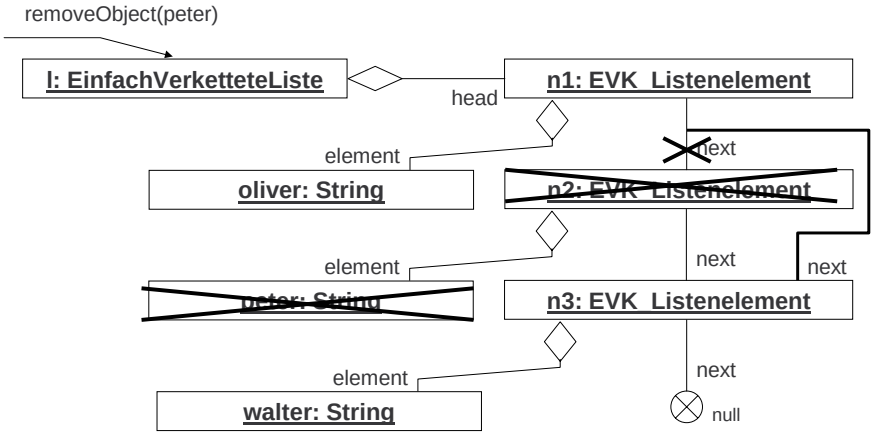
Implementierung (3/5): Ergänzung der einfach verketteten Liste um den Dienst isIn() (Java)

```
// Ergänzung Klassendefinition
// EinfachVerketteteListe
   (Manipulatoren)
   public bool isIn(Object o) {
      EVK_Listenelement a;
      a = head;
      while (a != null &&
               a.getElement() != o)
         a = a.getNext();
      if (a == null) return false;
      return true;
   } //...
```



Set (5/8)

Implementierung (4/5): removeObject() im Model



Set (6/8)

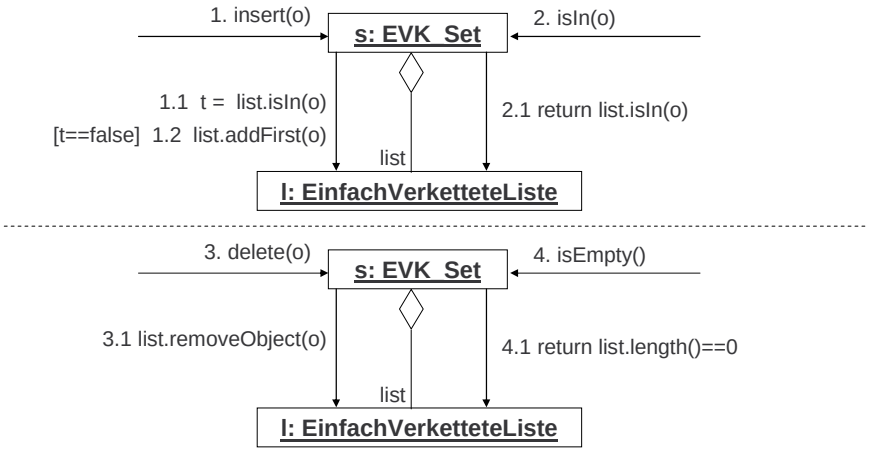
Implementierung (5/5): Ergänzung der einfach verketteten Liste um den Dienst removeObject() (Java)

```
// Ergänzung Klassendefinition
// EinfachVerketteteListe (Manipulatoren)
   public void removeObject(Object o) {
      // Achtung: Methode setzt
      // vorhandenes Objekt voraus!
      EVK_Listenelement a, v = null;
      a = head;
      while (a != null
            && a.getElement() != o) {
         v = a;
         a = a.getNext();
      }
      if (v == null) head = head.getNext();
      else v.setNext(a.getNext());
   } //...
```

Prinzip: Die Liste wird durchwandert, wobei immer zwei Knoten betrachtet werden: der aktuelle Knoten (a; zeigt am Ende der Schleife auf das gesuchte Objekt) und der Vorgängerknoten (v) von a (Ausnahme: o ist im ersten Knoten enthalten). Enthält a das Objekt, wird der Zeiger von v auf den a folgenden Knoten „umgelegt“.

Set (7/8)

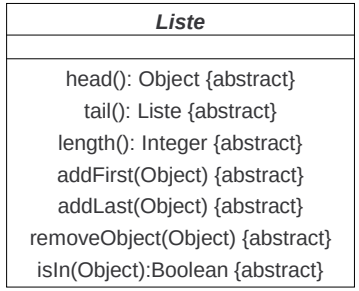
Implementierung (7/5): Kollaborationsdiagramme



Set (8/8)

Anmerkungen:

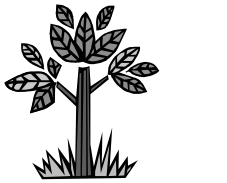
- Java-Implementierung analog zu Stack.
- Andere (bessere) Implementierungen sind möglich. Insb. wurden keine Fehlersituationen berücksichtigt.
- Auch die Dienste removeObject() und isIn() der Klasse EinfachVerketteteListe sind generell bei Listen sinnvoll und sollten als abstrakte Methoden in der Klasse Liste definiert sein:

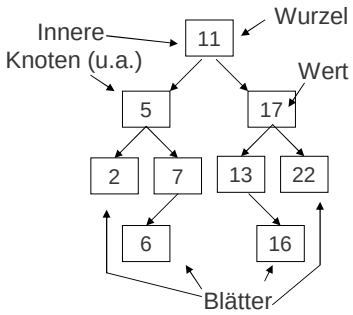


Binärbäume (1/29)

Bäume sind uns schon bei der Darstellung grammatikalischer Strukturen begegnet (Ableitungsbaum, s. o.). Weitere Verwendung finden sie z.B. bei der Verwaltung von Verzeichnisbäumen (Dateisystem).

- Elemente einer Baumstruktur werden auch als Knoten bezeichnet.
- Jeder Knoten trägt einen Wert (value).
- Jeder Knoten verweist auf maximal zwei sog. Kind-Knoten (auch Kinder).
- Der Ursprungsknoten eines Baumes wird als Wurzel (root) bezeichnet.
- Knoten ohne Kind-Knoten werden als Blätter, aller anderen als innere Knoten bezeichnet.





Binärbäume (2/29)

Definition des ADT Tree:

Anmerkungen:

- Bäume sind eine *nicht-lineare* Datenstruktur:

  Eine Implementierung über Listen ist (nicht gut) möglich.
- Bäume sind eine *rekursive* Datenstruktur

  Jeder Knoten kann *als Wurzel eines (Teil-)Baums* aufgefasst werden.

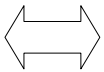
type Tree(T)
import Bool
operators
empty : → Tree
bin : Tree×T×Tree → Tree
left : Tree → Tree
right : Tree → Tree
value : Tree → T
empty? : Tree → Bool
axioms ∀ *t*: T, ∀ *x,y*: Tree
left(bin(*x,t,y*)) = *x*
right(bin(*x,t,y*)) = *y*
value(bin(*x,t,y*)) = *t*
empty?(empty) = *true*
empty?(bin(*x,t,y*)) = *false*

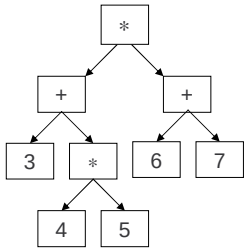
Binärbäume (3/29)

Anwendungsbeispiel:

Terme mit bis zu 2-stelligen Operationen (Operatorbäume oder Kantorowitsch-Bäume), z.B.

(3+(4∗5))∗(6+7)





Als Beispiele für Algorithmen auf Binärbäumen studieren wir die Übersetzungen

1. Term → Binärbaum
2. Binärbaum → Term

Zugrunde liegende Syntax (s.a. 2. Beispiel zu Kellerspeicher):

A ::= *B* =

B ::= Int | *B*+*B* | *B*∗*B* | (*B*)

Binärbäume (4/29)

Term → Binärbaum (1/4)

Der Algorithmus ist der selbe wie bei dem 2. Anwendungsbeispiel zum Stack. Wir verwenden wieder *zwei* Stacks:

1. S_1 für Operatoren und
2. S_2 für Binärbäume (anstelle Operanden).

An die Stelle der Termauswertung mit ⟦*x*∗*y*⟧ rückt nun die Baumkonstruktion.

Erinnerung Notation:

- $z_1 \ldots z_n S$ = push(. . . push(S, z_n),. . . ,z_1)
- ⟦*x*+*y*⟧ entspreche dem berechneten Wert *x*+*y*, ⟦*x*∗*y*⟧ entsprechend
- ⟦*x*⟧ entspreche dem Wert *des Integer-Zeichens oder -Worts* x

Erinnerung: Keller: Anwendungsbeispiel II (2/3)

Implementierung (funktional):

value(*t*)	= eval ⟨ *t*, empty, empty ⟩	// Start der Evaluation
eval ⟨ (*t*, S_1, S_2 ⟩	= eval ⟨ *t*, (S_1, S_2 ⟩	// Ablegen der geöffneten Klammer
eval ⟨ ∗*t*, S_1, S_2 ⟩	= eval ⟨ *t*, ∗S_1, S_2 ⟩	// Ablegen des '∗'-Operators
eval ⟨ +*t*, ∗S_1, *yx*S_2 ⟩	= eval ⟨ +*t*, S_1, ⟦*x*∗*y*⟧ S_2 ⟩	// Auswerten (Punkt vor Strich)
eval ⟨ +*t*, S_1, S_2 ⟩	= eval ⟨ *t*, +S_1, S_2 ⟩	// Ablegen des '+'-Operators
eval ⟨)*t*, +S_1, *yx*S_2 ⟩	= eval ⟨)*t*, S_1, ⟦*x*+*y*⟧ S_2 ⟩	// Auswerten letzte Addition vor ')'
eval ⟨)*t*, ∗S_1, *yx*S_2 ⟩	= eval ⟨)*t*, S_1, ⟦*x*∗*y*⟧ S_2 ⟩	// Auswerten letzte Multiplikation vor ')'
eval ⟨)*t*, (S_1, S_2 ⟩	= eval ⟨ *t*, S_1, S_2 ⟩	// Beenden einer „Teilauswertung“
eval ⟨ =, +S_1, *yx*S_2 ⟩	= eval ⟨ =, S_1, ⟦*x*+*y*⟧ S_2 ⟩	// Auswerten (letzte Addition rechts)
eval ⟨ =, ∗S_1, *yx*S_2 ⟩	= eval ⟨ =, S_1, ⟦*x*∗*y*⟧ S_2 ⟩	// Auswerten (letzte Multipl. rechts)
eval ⟨ =, empty, *x* empty ⟩ = *x*		// Ende der Auswertung erreicht
eval ⟨ ***x***t, S_1, S_2 ⟩	= eval ⟨ *t*, S_1, ⟦***x***⟧ S_2 ⟩	// Integer-*Zeichen* wird *als Wert* auf den Stack gelegt

Erinnerung Keller: Anwendungs-beispiel II (3/3)

Implementierung:

1 value(*t*) = eval ⟨ *t*, empty, empty ⟩
2 eval ⟨ (*t*, S_1, S_2 ⟩ = eval ⟨ *t*, (S_1, S_2 ⟩
3 eval ⟨ ∗*t*, S_1, S_2 ⟩ = eval ⟨ *t*, ∗S_1, S_2 ⟩
4 eval ⟨ +*t*, ∗S_1, *yx*S_2 ⟩ = eval ⟨ +*t*, S_1, ⟦*x*∗*y*⟧ S_2 ⟩
5 eval ⟨ +*t*, S_1, S_2 ⟩ = eval ⟨ *t*, +S_1, S_2 ⟩
6 eval ⟨)*t*, +S_1, *yx*S_2 ⟩ = eval ⟨)*t*, S_1, ⟦*x*+*y*⟧ S_2 ⟩
7 eval ⟨)*t*, ∗S_1, *yx*S_2 ⟩ = eval ⟨)*t*, S_1, ⟦*x*∗*y*⟧ S_2 ⟩
8 eval ⟨)*t*, (S_1, S_2 ⟩ = eval ⟨ *t*, S_1, S_2 ⟩
9 eval ⟨ =, +S_1, *yx*S_2 ⟩ = eval ⟨ =, S_1, ⟦*x*+*y*⟧ S_2 ⟩
A eval ⟨ =, ∗S_1, *yx*S_2 ⟩ = eval ⟨ =, S_1, ⟦*x*∗*y*⟧ S_2 ⟩
B eval ⟨ =, empty, *x* empty ⟩ = *x*
C eval ⟨ ***x***t, S_1, S_2 ⟩ = eval ⟨ *t*, S_1, ⟦***x***⟧ S_2 ⟩

bearbeiteter Term *t*	Stack S_1	Stack S_2	Regel
(3+4∗5)∗(6+7)=	empty	empty	2
3+4∗5)∗(6+7)=	(	empty	C
+4∗5)∗(6+7)=	(	3	5
4∗5)∗(6+7)=	+ (	3	C
∗5)∗(6+7)=	+ (	4 3	3
5)∗(6+7)=	∗ + (	4 3	C
)∗(6+7)=	∗ + (	5 4 3	7
)∗(6+7)=	+ (	20 3	6
)∗(6+7)=	(	23	8
∗(6+7)=	empty	23	3
(6+7)=	∗	23	2
6+7)=	(∗	23	C
+7)=	(∗	6 23	5
7)=	+ (∗	6 23	C
)=	+ (∗	7 6 23	6
)=	(∗	13 23	8
=	∗	13 23	A
=	empty	299	B

Binärbäume (5/29)

Term → Binärbaum (2/4)

Implementierung (funktional):

value(*t*)	= eval ⟨ *t*, empty, empty ⟩	// Start der Evaluation
eval ⟨ (*t*, S_1, S_2 ⟩	= eval ⟨ *t*, (S_1, S_2 ⟩	// Ablegen d. geöffneten Klammer
eval ⟨ ∗*t*, S_1, S_2 ⟩	= eval ⟨ *t*, ∗S_1, S_2 ⟩	// Ablegen des '∗'-Operators
eval ⟨ +*t*, ∗S_1, *yx*S_2 ⟩	= eval ⟨ +*t*, S_1, *bin*(*x*,∗,*y*)S_2 ⟩	// Teilbaumkonstruktion (TK)
eval ⟨ +*t*, S_1, S_2 ⟩	= eval ⟨ *t*, +S_1, S_2 ⟩	// Ablegen des '+'-Operators
eval ⟨)*t*, +S_1, *yx*S_2 ⟩	= eval ⟨)*t*, S_1, *bin*(*x*,+,*y*)S_2 ⟩	// TK letzte Addition vor ')'
eval ⟨)*t*, ∗S_1, *yx*S_2 ⟩	= eval ⟨)*t*, S_1, *bin*(*x*,∗,*y*)S_2 ⟩	// TK letzte Mult. vor ')'
eval ⟨)*t*, (S_1, S_2 ⟩	= eval ⟨ *t*, S_1, S_2 ⟩	// Beenden e. „Teilauswertung“
eval ⟨ =, +S_1, *yx*S_2 ⟩	= eval ⟨ =, S_1, *bin*(*x*,+,*y*)S_2 ⟩	// TK letzte Addition rechts
eval ⟨ =, ∗S_1, *yx*S_2 ⟩	= eval ⟨ =, S_1, *bin*(*x*,∗,*y*)S_2 ⟩	// TK letzte Multipl. rechts
eval ⟨ =, empty, *x* empty ⟩ = *x*		// Ende der Konstruktion erreicht
eval ⟨ ***x***t, S_1, S_2 ⟩	= eval ⟨ *t*, S_1, *bin*(*empty*, ***x***,*empty*)S_2 ⟩	// Int.-*Zeichen* wird *als Blatt* auf den Stack gelegt

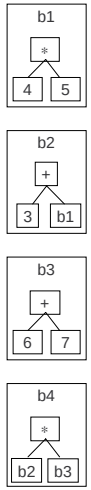
Binärbäume (6/29)

Term → Binärbaum (3/4)

Implementierung:

1 value(*t*) = eval ⟨ *t*, empty, empty ⟩
2 eval ⟨ (*t*, S_1, S_2 ⟩ = eval ⟨ *t*, (S_1, S_2 ⟩
3 eval ⟨ ∗*t*, S_1, S_2 ⟩ = eval ⟨ *t*, ∗S_1, S_2 ⟩
4 eval ⟨ +*t*, ∗S_1, *yx*S_2 ⟩ = eval ⟨ +*t*, S_1, *bin*(*x*,∗,*y*)S_2 ⟩
5 eval ⟨ +*t*, S_1, S_2 ⟩ = eval ⟨ *t*, +S_1, S_2 ⟩
6 eval ⟨)*t*, +S_1, *yx*S_2 ⟩ = eval ⟨)*t*, S_1, *bin*(*x*,+,*y*)S_2 ⟩
7 eval ⟨)*t*, ∗S_1, *yx*S_2 ⟩ = eval ⟨)*t*, S_1, *bin*(*x*,∗,*y*)S_2 ⟩
8 eval ⟨)*t*, (S_1, S_2 ⟩ = eval ⟨ *t*, S_1, S_2 ⟩
9 eval ⟨ =, +S_1, *yx*S_2 ⟩ = eval ⟨ =, S_1, *bin*(*x*,+,*y*)S_2 ⟩
A eval ⟨ =, ∗S_1, *yx*S_2 ⟩ = eval ⟨ =, S_1, *bin*(*x*,∗,*y*)S_2 ⟩
B eval ⟨ =, empty, *x* empty ⟩ = *x*
C eval ⟨ ***x***t, S_1, S_2 ⟩ = eval ⟨ *t*, S_1, *bin*(*empty*, ***x***,*empty*)S_2 ⟩

bearbeiteter Term *t*	Stack S_1	Stack S_2	Regel
(3+4∗5)∗(6+7)=	empty	empty	2
3+4∗5)∗(6+7)=	(	empty	C
+4∗5)∗(6+7)=	(	3	5
4∗5)∗(6+7)=	+ (	3	C
∗5)∗(6+7)=	+ (	4 3	3
5)∗(6+7)=	∗ + (	4 3	C
)∗(6+7)=	∗ + (	5 4 3	7
)∗(6+7)=	+ (	b1 3	6
)∗(6+7)=	(	b2	8
∗(6+7)=	empty	b2	3
(6+7)=	∗	b2	2
6+7)=	(∗	b2	C
+7)=	(∗	6 b2	5
7)=	+ (∗	6 b2	C
)=	+ (∗	7 6 b2	6
)=	(∗	b3 b2	8
=	∗	b3 b2	A
=	empty	b4	B



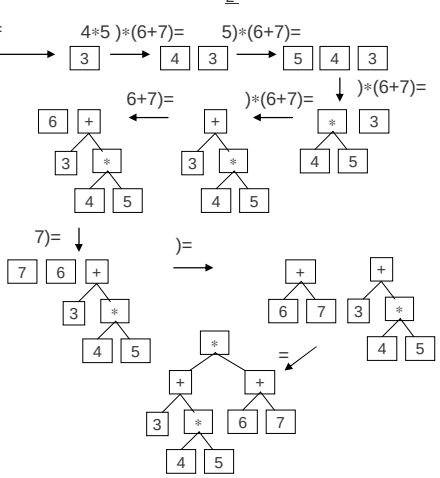
Binärbäume (7/29)

Term → Binärbaum (4/4)

bearbeiteter Term *t*

(3+4∗5)∗(6+7)=

Stack S_2 (Top links)



Implementierung:

1 value(*t*) = eval ⟨ *t*, empty, empty ⟩
2 eval ⟨ (*t*, S_1, S_2 ⟩ = eval ⟨ *t*, (S_1, S_2 ⟩
3 eval ⟨ ∗*t*, S_1, S_2 ⟩ = eval ⟨ *t*, ∗S_1, S_2 ⟩
4 eval ⟨ +*t*, ∗S_1, *yx*S_2 ⟩ = eval ⟨ +*t*, S_1, *bin*(*x*,∗,*y*)S_2 ⟩
5 eval ⟨ +*t*, S_1, S_2 ⟩ = eval ⟨ *t*, +S_1, S_2 ⟩
6 eval ⟨)*t*, +S_1, *yx*S_2 ⟩ = eval ⟨)*t*, S_1, *bin*(*x*,+,*y*)S_2 ⟩
7 eval ⟨)*t*, ∗S_1, *yx*S_2 ⟩ = eval ⟨)*t*, S_1, *bin*(*x*,∗,*y*)S_2 ⟩
8 eval ⟨)*t*, (S_1, S_2 ⟩ = eval ⟨ *t*, S_1, S_2 ⟩
9 eval ⟨ =, +S_1, *yx*S_2 ⟩ = eval ⟨ =, S_1, *bin*(*x*,+,*y*)S_2 ⟩
A eval ⟨ =, ∗S_1, *yx*S_2 ⟩ = eval ⟨ =, S_1, *bin*(*x*,∗,*y*)S_2 ⟩
B eval ⟨ =, empty, *x* empty ⟩ = *x*
C eval ⟨ **x***t*, S_1, S_2 ⟩
= eval ⟨ *t*, S_1, *bin*(*empty*, **x**,*empty*)S_2 ⟩

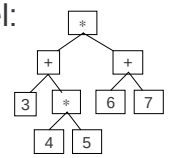
Binärbäume (8/29)

Binärbaum → Term (1/5)

Der folgende sehr einfache funktionale Algorithmus erzeugt einen vollständig geklammerten Term in (der traditionellen) Infix-Schreibweise.

(1) infixTerm(empty) = [ ]
(2) infixTerm(*bin*(*x*,∗,*y*)) = [(] ++ infixTerm(*x*) ++ [∗] ++ infixTerm(*y*) ++ [)]
(3) infixTerm(*bin*(*x*,+,*y*)) = [(] ++ infixTerm(*x*) ++ [+] ++ infixTerm(*y*) ++ [)]
(4) infixTerm(*bin*(*empty*,*x*,*empty*)) = *x*

Beispiel:

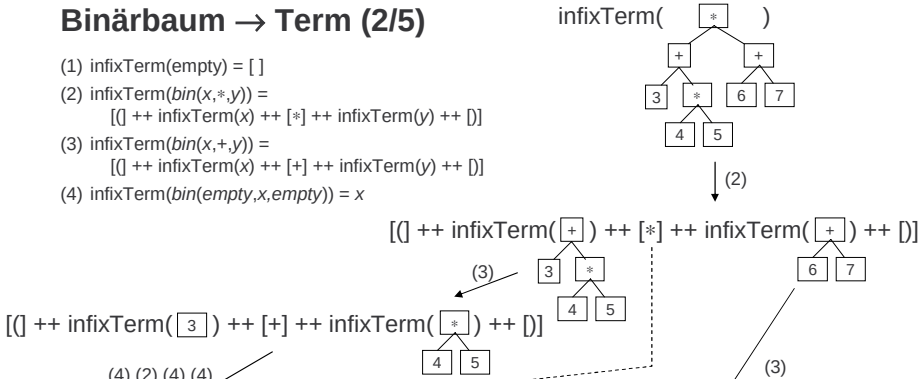


"(3+(4∗5))∗(6+7)"

Binärbäume (9/29)

Binärbaum → Term (2/5)

(1) infixTerm(empty) = [ ]
(2) infixTerm(*bin*(*x*,∗,*y*)) =
[(] ++ infixTerm(*x*) ++ [∗] ++ infixTerm(*y*) ++ [)]
(3) infixTerm(*bin*(*x*,+,*y*)) =
[(] ++ infixTerm(*x*) ++ [+] ++ infixTerm(*y*) ++ [)]
(4) infixTerm(*bin*(*empty*,*x*,*empty*)) = *x*



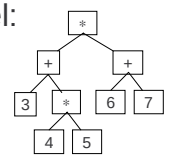
Binärbäume (10/29)

Binärbaum → Term (3/5)

Der folgende funktionale Algorithmus erzeugt einen klammerlosen Term in Präfix-Schreibweise.

(1) präfixTerm(empty) = [ ]
(2) präfixTerm(*bin*(*x*,∗,*y*)) = [∗] ++ präfixTerm(*x*) ++ präfixTerm(*y*)
(3) präfixTerm(*bin*(*x*,+,*y*)) = [+] ++ präfixTerm(*x*) ++ präfixTerm(*y*)
(4) präfixTerm(*bin*(*empty*,*x*,*empty*)) = *x*

Beispiel:

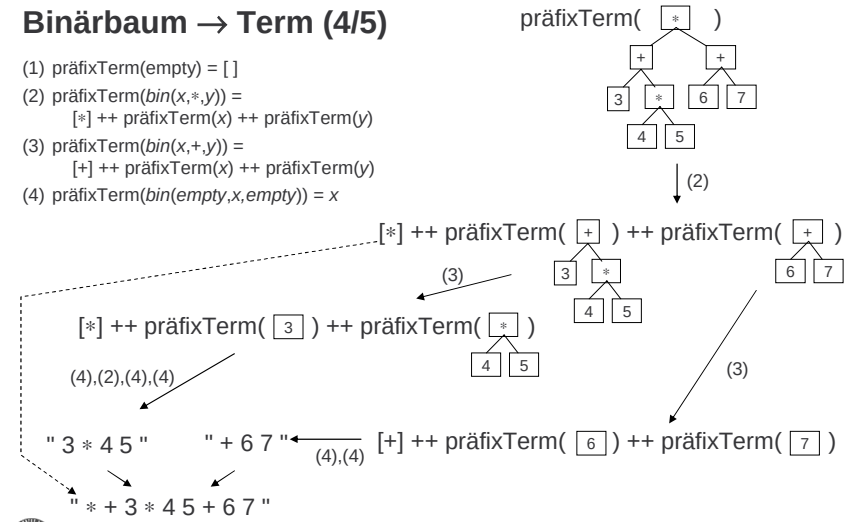


" ∗ + 3 ∗ 4 5 + 6 7"

Binärbäume (11/29)

Binärbaum → Term (4/5)

(1) präfixTerm(empty) = [ ]
(2) präfixTerm(*bin*(*x*,∗,*y*)) =
[∗] ++ präfixTerm(*x*) ++ präfixTerm(*y*)
(3) präfixTerm(*bin*(*x*,+,*y*)) =
[+] ++ präfixTerm(*x*) ++ präfixTerm(*y*)
(4) präfixTerm(*bin*(*empty*,*x*,*empty*)) = *x*



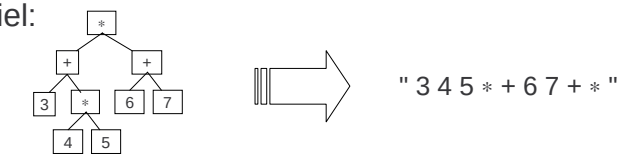
Binärbäume (12/29)

Binärbaum → Term (5/5)

Der folgende funktionale Algorithmus erzeugt einen klammerlosen Term in Postfix-Schreibweise.

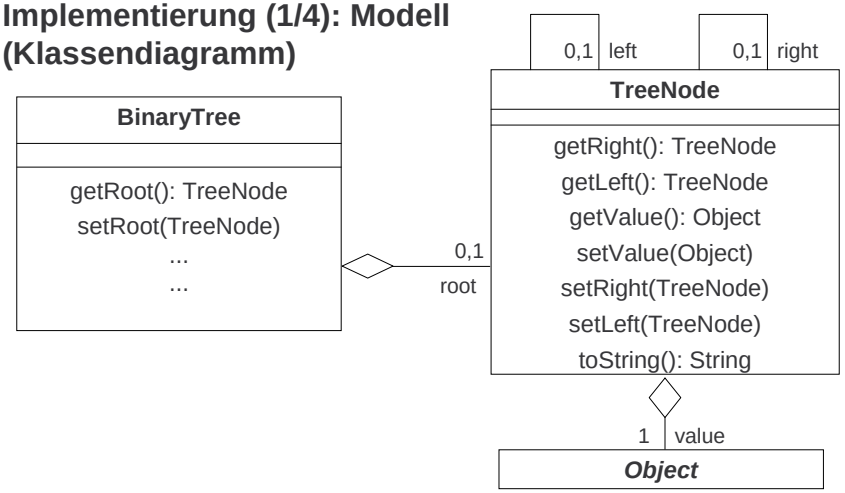
(1) postfixTerm(empty) = [ ]
(2) postfixTerm(*bin*(*x*,∗,*y*)) = postfixTerm(*x*) ++ postfixTerm(*y*) ++ [∗]
(3) postfixTerm(*bin*(*x*,+,*y*)) = postfixTerm(*x*) ++ postfixTerm(*y*) ++ [+]
(4) postfixTerm(*bin*(*empty*,*x*,*empty*)) = *x*

Beispiel:



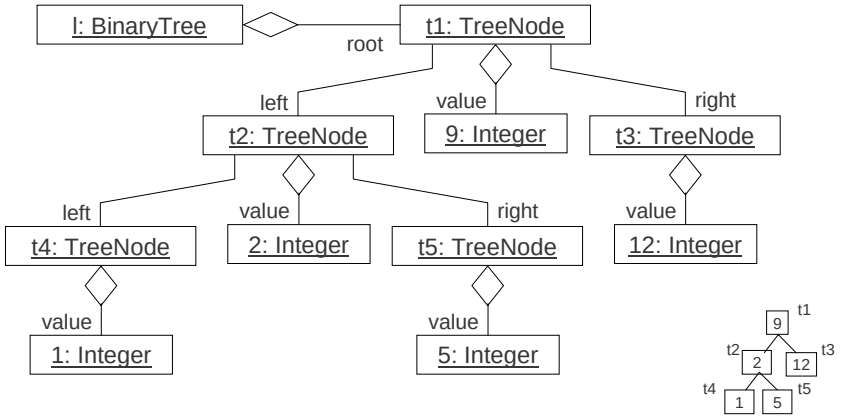
Binärbäume (13/29)

Implementierung (1/4): Modell (Klassendiagramm)



Binärbäume (14/29)

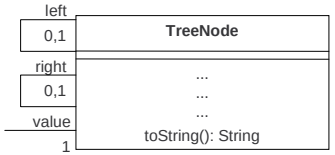
Implementierung (2/4): Binärbaum (Objektmodell)



Binärbäume (15/29)

Implementierung (3/4): Java (Auszugsweise)

```
public class TreeNode
{
//Attributdeklarationen
    protected Object value;
    protected TreeNode left,right;
//Konstruktoren
    public TreeNode(Object o) {
          value = o; }
// Selektoren
    public String toString () {
          return value.toString();
    } // Standard (>Sollte von jeder Klasse
    // überschrieben werden (Polymor.))
```



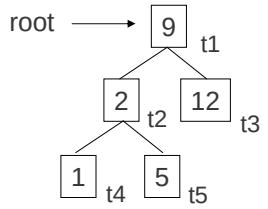

```
// Selektoren
    public Object getValue() {
        return value;
    }
    public TreeNode getLeft() {
        return left;
    }       //...Manipulatoren...
}
```


Binärbäume (16/29)

Implementierung (4/4): Verwendung (Java)

Aufbau eines Binärbaumes (Werte: Zeichenketten):

```
TreeNode t5 = new TreeNode(5);
TreeNode t4 = new TreeNode(1);
TreeNode t2 = new TreeNode(2);
t2.setLeft(t4); t2.setRight(t5);
TreeNode t3 = new TreeNode(12);
TreeNode t1 = new TreeNode(9);
t1.setLeft(t2); t1.setRight(t3);
Tree t = new Tree();
t.setRoot(t1)
```



Achtung: Es werden int-Zahlen verwendet, *obwohl* int-Zahlen in Java *keine Objekte* im eigentlichen Sinne sind (wie von der Klassendefinition von TreeNode als Typ des Attributs value gefordert) sondern Werte eines primitiven Datentyps und erst in Objekte der Klasse Integer umgewandelt werden müssten.

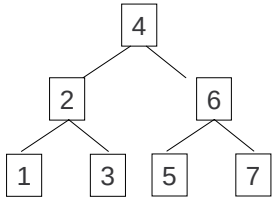
Binärbäume (17/29)

Traversierung (Durchwandern) eines Binärbaumes (1/4)

Es gibt verschiedene Varianten, einen Binärbaum zu durchwandern (und die Knotenwerte zu bearbeiten):

Zur sog. Tiefensuche gehören:

- Inorder: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$
- Preorder: $4 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 5 \rightarrow 7$
- Postorder: $1 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 6 \rightarrow 4$



Der Breitensuche entspricht:

- Levelorder: $4 \rightarrow 2 \rightarrow 6 \rightarrow 1 \rightarrow 3 \rightarrow 5 \rightarrow 7$

Anmerkung: Der Baum wird bei der Tiefensuche immer rekursiv von links nach rechts durchwandert. Der Unterschied liegt im Zeitpunkt der Notierung (bzw. Verarbeitung) des Wertes eines Knotens ((zwischen, vor oder nach den Aufrufen).

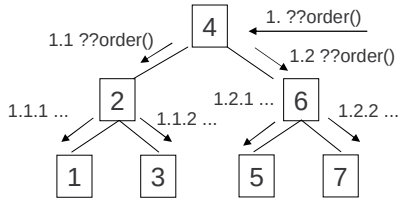
Binärbäume (18/29)

Traversierung (Durchwandern) eines Binärbaumes (2/4)

Algorithmenschemata (in-/pre- und postorder):

Als Methoden der Klasse BinaryTree:

```
public void inorder (TreeNode k) {
 if (k!=null) {
   this.inorder(k.getLeft());
   this.verarbeiteWert(k.getValue());
   this.inorder(k.getRight()); }
}
public void preorder (TreeNode k) {
 if (k!=null) {
   this.verarbeiteWert(k.getValue());
   this.preorder(k.getLeft());
   this.preorder(k.getRight()); } }
```




```
public void postorder (TreeNode k) {
 if (k!=null) {
    this.postorder(k.getLeft());
    this.postorder(k.getRight());
    this.verarbeiteWert(k.getValue()); }
}
```

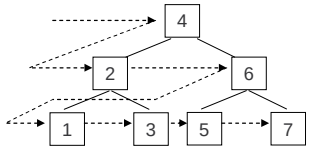
Binärbäume (19/29)

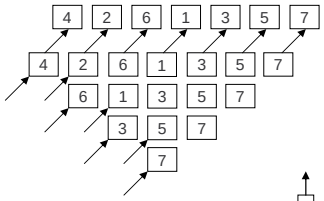
Traversierung (Durchwandern) eines Binärbaumes (3/4)

Algorithmenschema für Levelorder:

Levelorder wird nicht rekursiv, sondern mittels einer Queue als Methode der Klasse BinaryTree realisiert:

```
public void levelorder () {
    TreeNode n;
    EVK_Queue q = new EVK_Queue();
    q.enter(root);
    while (! q.isEmpty()) {
       n = (TreeNode) q.front(); q.leave();
       this.verarbeiteWert(n.getValue());
       if (n.getLeft() != null) q.enter(n.getLeft());
       if (n.getRight() != null) q.enter(n.getRight());
    } }
```





Entwicklung der Queue über die Zeit

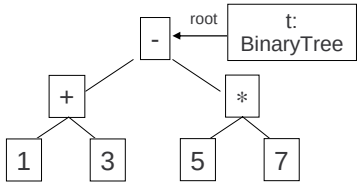
Binärbäume (20/29)

Traversierung (Durchwandern) eines Binärbaumes (4/4)

Anwendungsbeispiel: Textuelle Inorder-Ausgabe eines Binärbaums

Als Methode der Klasse BinaryTree:

```
protected void printInorder (TreeNode k) {
   if (k!=null) {
       this.printInorder (k.getLeft());
       System.out.println(n.toString());
         //Verarbeitung = Aufruf der
         //Standardfunktion zur Textausgabe
       this.printInorder (k.getRight()); }
}
public void printInorder() {
   this.printInorder(root);
}
```



Anmerkungen:

- Angewendet auf den Binary-Tree t führt t.printInorder() zur Ausgabe "1+3-5*7"
- Implementierung von printPostorder() und printPreorder() analog.

Binärbäume (21/29)

Suchbäume (1/10)

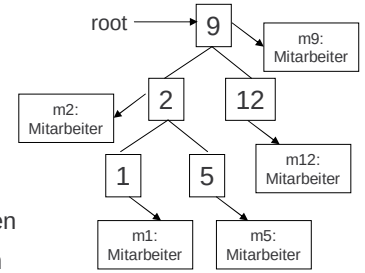
Neben der Verwendung als Datenstruktur zur hierarchischen Organisation von Objekten werden Bäume auch zur Unterstützung einer effizienten Suche nach Objekten eingesetzt.

Grundprinzip:

– Wert eines Knotens ist ein identifizierender *Schlüsselwert* aus einer *geordneten* Grundmenge (totale Ordnung ≤ ist Voraussetzung): z.B. Nachname, Personalnummer

– Knoten enthält zusätzlichen *Verweis* auf „Nutzdaten“: z.B. Personendaten mit Adressen

– Anordnung der Wert/Verweis-Paare im Baum spiegelt Ordnung wieder => Einfügen und Löschen von Knoten i.d.R. kompliziert



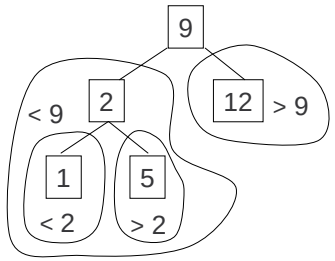
Binärbäume (21/29)

Suchbäume (2/10): Spezialfall binäre Suchbäume

Definition:

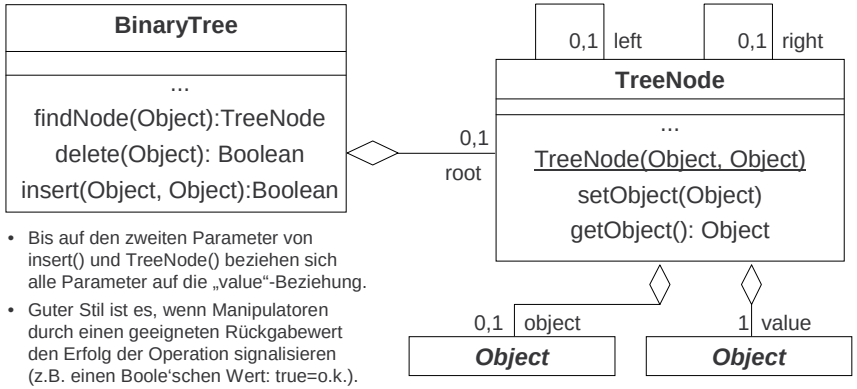
Ein *binärer Suchbaum* ist ein Binärbaum, bei dem für alle Knoten bin(b_1,v,b_2) gilt: für jeden Knoten v_1 in b_1 ist $v_1 < v$, und für jeden Knoten v_2 in b_2 ist $v_2 > v$

Beispiel:



Binärbäume (22/29)

Suchbäume (3/10): Objektorientierte Modellierung binärer Suchbäume (Implementierung) als Erweiterung der Klasse BinaryTree (>Unterscheidung des Wertes vom referenzierten Objekt des Knotens).



- Bis auf den zweiten Parameter von insert() und TreeNode() beziehen sich alle Parameter auf die „value“-Beziehung.
- Guter Stil ist es, wenn Manipulatoren durch einen geeigneten Rückgabewert den Erfolg der Operation signalisieren (z.B. einen Boole'schen Wert: true=o.k.).

Binärbäume (23/29)

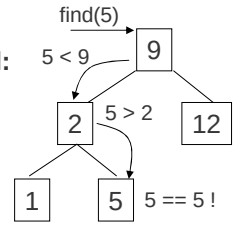
Suchbäume (4/10): Suchen in binären Suchbäume

Algorithmenschema (Java-Pseudocode):

Iterative Lösung (als Methode der Klasse BinaryTree):

```
public TreeNode findNode (Object v) {
    TreeNode n = root;
    while (n != null) {
        if (n.getValue() == v) return n;
        if (v<n.getValue()) n = n.getLeft();
        else n = n.getRight();
    }
    return null;  //Wert nicht vorhanden
}
```

Beispiel:



Anmerkungen:

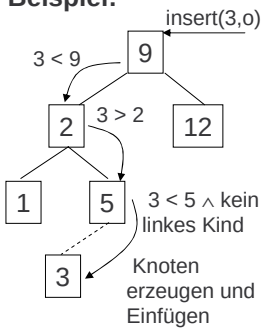
- Der Dienst lässt sich auch elegant rekursiv formulieren (als Dienst der Klasse BinaryTree oder TreeNode > Übung)
- Der Vergleich "<" ist i.A. in Java für die betrachteten Klassen durch einen eigenen Dienst zu realisieren.

Binärbäume (24/29)

Suchbäume (5/10): Einfügen in binäre Suchbäume

Wichtig: Erhalten der Eigenschaft des binären Suchbaumes

Beispiel:



Methode (Pseudocode) der Klasse BinaryTree:

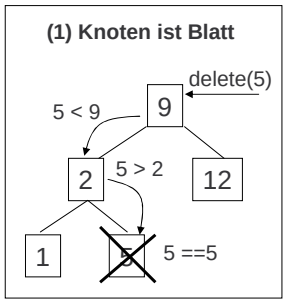
```
public boolean insert (Object v, Object o) {
    TreeNode parent = null, child = root, newNode;
    while (child != null) {
        parent = child;
        if (child.getValue() == v) return false;
        if (v<child.getValue()) child = child.getLeft();
        else child = child.getRight();
    }
    newNode = new TreeNode(v,o);
    if (parent == null) root = newNode;
    else    if (v<parent.getValue())
                parent.setLeft(newNode);
            else parent.setRight(newNode);
    return true; }
```

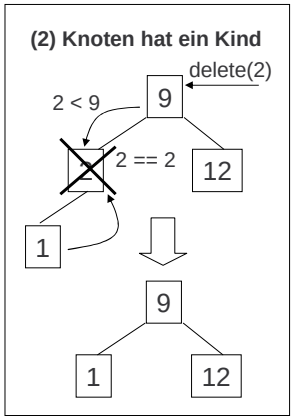

Binärbäume (25/29)

Suchbäume (6/10): Löschen in binären Suchbäumen

Wichtig: Erhalten der Eigenschaft des binären Suchbaumes

3 Fälle (Beispiele):

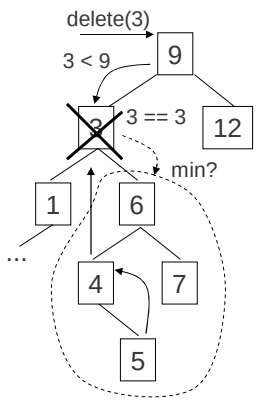




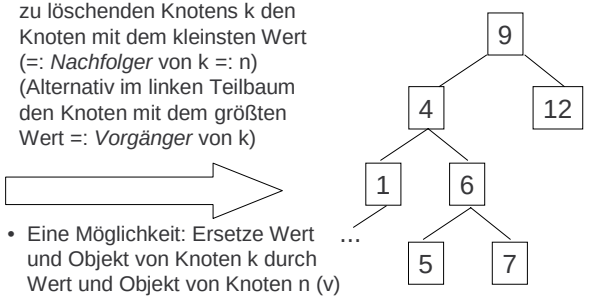
Binärbäume (26/29)

Suchbäume (7/10): Löschen in binären Suchbäumen (Fortsetzung)

(3) Knoten hat zwei Kinder



- Suche im rechten Teilbaum des zu löschenden Knotens k den Knoten mit dem kleinsten Wert (=: *Nachfolger* von k =: n) (Alternativ im linken Teilbaum den Knoten mit dem größten Wert =: *Vorgänger* von k)



- Eine Möglichkeit: Ersetze Wert und Objekt von Knoten k durch Wert und Objekt von Knoten n (v)
- Lösche Knoten n (> Einstufige Rekursion)

Anmerkung: n ist gleichzeitig der minimale Wert im rechten Teilbaum bzw. v der maximale im linken.

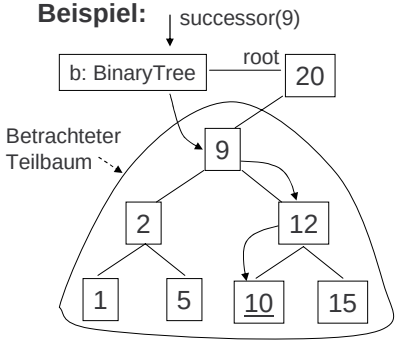
Binärbäume (27/29)

Suchbäume (8/10): Löschen in binären Suchbäumen (Fortsetzung)

Implementierung der Löschfunktion (1/3): Zunächst Hilfsmethode der Klasse BinaryTree zum Finden des Nachfolgers (successor) der Wurzel eines Teilbaums. Methoden (Pseudocode) :

```
protected TreeNode subtreeSuccessor
    (TreeNode t) {
    // Beginn mit dem rechten Kind
    TreeNode c = t.getRight();
    if (c== null) return null;
    // den Baum nach links durchwandern
    while (c.getLeft() != null)
        c = c.getLeft();
    return c;
}
```

Beispiel:



Binärbäume (28/29)

Suchbäume (9/10): Löschen in binären Suchbäumen (Fortsetzung)

Implementierung der Löschfunktion (2/3): Methode der Klasse BinaryTree (Pseudocode) :

```
public boolean delete (Object v) {
    TreeNode parent = null, child,
    node = root, succ;
    // 1. zunächst den Knoten suchen
    while (node != null
        && v != node.getValue() ) {
        parent = node;
        if (v< node.getValue())
          node = node.getLeft();
          else node = node.getRight();
    }
    // kein Knoten gefunden
    if (node ==null) return false;
    // 2. Den Knoten löschen
    // 2.1 Der Knoten ist die Wurzel
    if (parent == null) {
        if (node.getLeft()==null)
            root = node.getRight();
        else if (node.getRight()==null)
            root = node.getLeft();
        else {
            succ = subtreeSuccessor(node);
            this.delete(succ.getValue());
            root.setValue(succ.getValue());
            root.setObject(succ.getObject());
        }
    } //...
```


Binärbäume (29/29)

Suchbäume (10/10): Löschen in binären Suchbäumen (Fortsetzung)

Implementierung der Löschfunktion (3/3): (Fortsetzung)

```
    // 2.2 Der Knoten ist nicht die Wurzel
    else if (node.getLeft() == null) {
    // und hat keinen linken Nachfolger
        if (parent.getLeft() == node)
            parent.setLeft(node.getRight());
        else parent.setRight(node.getRight());
    }
    else if (node.getRight() == null) {
    // bzw. keinen rechten Nachfolger
        if (parent.getLeft() == node)
            parent.setLeft(node.getLeft());
        else parent.setRight(node.getLeft());
    }
    else {
    // Knoten ist innerer Knoten
    succ = successor(node);
    this.delete(succ.getValue());
    node.setValue(succ.getValue());
    node.setObject(succ.getObject());
    }
    return true;
}
```

Anmerkungen:

- Voraussetzung bei dieser Variante ist, dass jeder Wert nur einmal vorkommt. Warum?
- Beim Löschen innerer Knoten muss das Löschen des „Ersatzknotens“ vor dem Kopieren von Wert und Objekt erfolgen! Warum?

2. Interne Realisierung & Pointer

Grundstruktur: Variable x vom Typ *T*

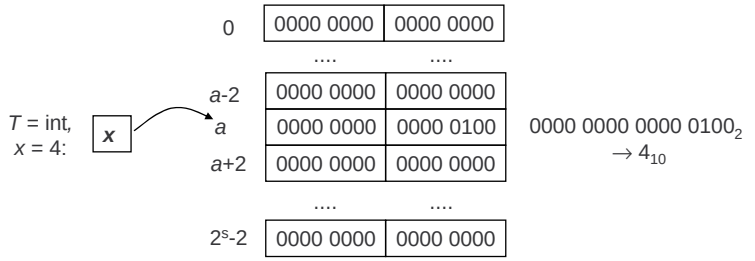
Interne Realisierung ≈ Abbildung der Variable bzw. des Variableninhalts in Speicherblock (RAM)

Primitive Datentypen (1/2)

- T = {bool, int, long, float, char, ...}
- Codierung des Wertebereichs erfordert *n* Bit (i.d.R. $n = m*8$} Beispiele (für Bitanzahl):
  - bool: n=1, char: n=8
  - int: n=16 (2 Byte: Wertebereich von -32769 bis 32768)
  - long: n=32 (4 Byte: Wertebereich von $-(2^{31}+1)$ bis 2^{31})
  - etc.

Primitive Datentypen (2/2)

- Variable wird auf Speicherblock abgebildet, deren Inhalt *direkt* als Variableninhalt interpretiert wird. Die Variable „zeigt“ auf diesen Bereich.



Speicher des Computers zur Laufzeit (Größe: 2^s Byte f. $s \in$ IN)
(a = Adresse des Variableninhalt(block)s von Variable x;
Größe des Blocks wird aus Typ abgeleitet; bei int 2 Byte)

Komplexe Datentypen / Klassenstrukturen (1/2)

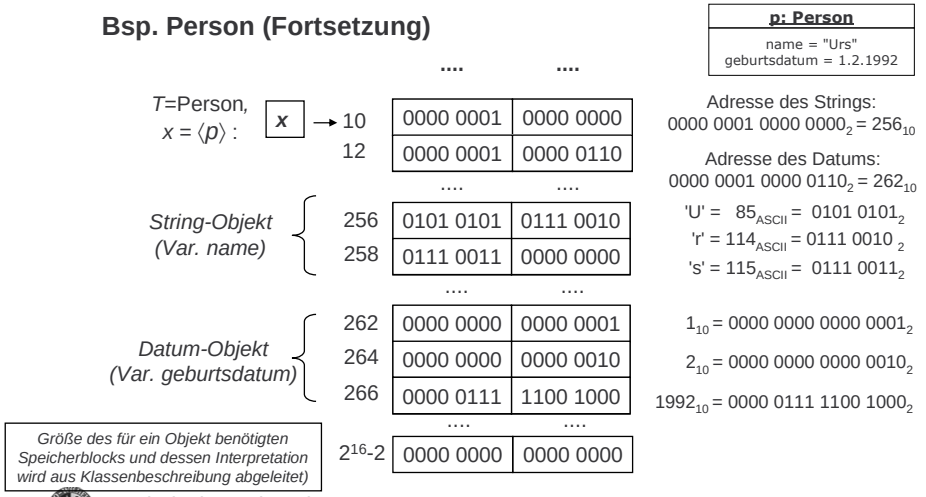
Bsp. Person

Vereinfachende Annahmen:

- Speicher habe die Größe von 2^{16} Byte
  => es genügen 2 Byte zur Kodierung der Adresse (0 bis 2^{16}-1) einer Speicherstelle (das erste Byte sei das höherwertige, das zweite das niederwertige)
- Ein String wird als Folge von Zeichen (char) in aufsteigenden Speicherstellen gespeichert. Das Ende eines Strings wird mit einem definierten Zeichen angegeben (hier: 0).
- Ein Datum wird durch drei int-Attribute (day, month, year) modelliert.

Komplexe Datentypen / Klassenstrukturen (2/2)

Bsp. Person (Fortsetzung)



Zeiger (Pointer: 1/2)

- Die von Variablen adressierten Speicherbereiche bzw. die Speicheradressen der Blöcke sind in verschiedenen Programmiersprachen direkt zugänglich, z.B. in C, C++, Pascal, Modula. Mit diesen Adressen kann "gearbeitet" werden (s.u.).
  Vorteil: Hardwarenahe Programmierung wird vereinfacht/möglich.
- In anderen sind diese sog. *Zeiger* "versteckt" (z.B. Java, Smalltalk), d.h. es wird implizit mit ihnen gearbeitet.
  Vorteile: Reduzierung von Fehlermöglichkeiten und "problemnahe" (von Hardware losgelöste) Entwicklung.

  Bsp. (Java): String str = new String("Peter");
  - Die Anweisung erzeugt eine "Zeiger-Variable" auf einen zunächst undefinierten Speicherbereich.
  - Die new-Anweisung (gefolgt vom Konstruktor) erzeugt dann ein Objekt, d.h. belegt einen entsprechenden Speicherbereich.
  - Die Zuweisung bewirkt dann ein Speichern der (nicht im Programm auslesbaren) Adresse dieses Blockes in der Variablen str.

Zeiger (Pointer: 2/2)

- Pragmatik explizit zugänglicher Zeiger:
  Aufbau von Datenstrukturen beliebiger Struktur und Größe. Damit verbunden ist i.d.R. die Notwendigkeit Speicherplatz im Programm zu reservieren (allokieren) und wieder freizugeben.
- Deklaration (Bsp.-Notation): var intzeiger: ↑int; // zunächst = null
  => Erzeugt eine „Zeiger-Variable“ für int-Objekte/Speicherbereiche
- Verwendung (Anschaulich mit beispielhafter Notation):
  - Speicherallokation (über Systemfunktion new()): new(intzeiger);
    => Reserviert int-Speicherblock und belegt intzeiger mit "Startadresse" des Blocks.
  - Zugriff auf Zeiger (Adresse): intzeiger := intzeiger + 2;
    => Addiert zwei Blockgrößen (bei int nach obiger Festlegung z.B. 2∗2 Speicherzellen)
  - Zugriff a. Var.-inhalt: intzeiger↑ := 4; var int intInhalt := intzeiger↑;
  - Speicherfreigabe: delete(intzeiger)
    => Variable intzeiger bleibt verwendbar!