



# Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 7: Referenzen und Zuweisungen



Wie werden Objekte "gemacht" ?  
(Mehr Details)

Wie modifizieren wir ihre Inhalte?

- Zuweisungen
- Speicherverwaltung, GC
- Mehr über Referenzen, Entitäten, Typen, Variablen...

# Referenzen sind schwierig...

---



require

-- "Meine Schwester ist ledig"  
-- "Meine Schwester ist in Dietlikon"

do

-- "Der Cousin meines Nachbarn heiratete  
-- gestern eine Ärztin in Sydney"

ensure

*(a doctor)*

-- "Meine Schwester ist ledig"  
-- "Meine Schwester ist in Dietlikon"

end

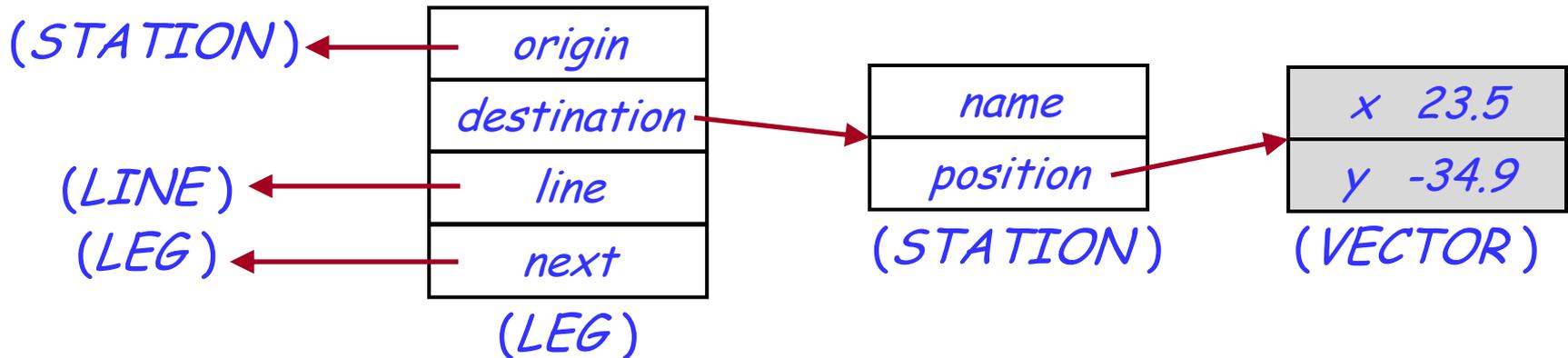


- Zur Laufzeit: Objekte (Softwaremaschinen)
- Im Programmtext: Klassen

Jede Klasse beschreibt eine Menge von möglichen Laufzeitobjekten

Ein Objekt besteht aus **Feldern**  
Jedes Feld ist ein **Wert**, entweder:

- Ein **Basiswert**: Ganze Zahl (*integer*), Zeichen (*character*), "reelle" Zahl, ...  
(genannt „expandierte Werte“)
- Eine **Referenz** auf ein anderes Objekt

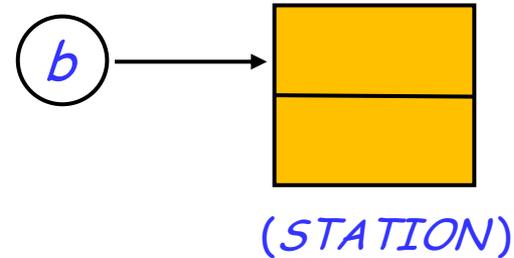


# Zwei Arten von Typen



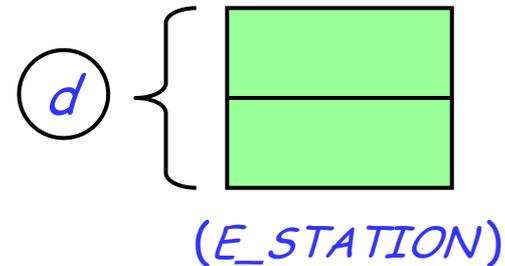
Referenztypen: Der Wert jeder Entität ist eine Referenz

*b: STATION*



Expandierte Typen: Der Wert jeder Entität ist ein Objekt

*d: E\_STATION*



# Expandierte und Referenztypen



Was ist der Unterschied zwischen:

➤ „Jeder Wagen hat einen Motor“



➤ „Jeder Wagen hat einen Hersteller“



?



Eine Klasse kann als expandiert deklariert werden:

*expanded class E\_STATION*

*... Der Rest wie in STATION ...*

Dann hat jede Entität deklariert als

*d: E\_STATION*

die eben beschriebene expandierte Semantik.

# Basistypen als expandierte Klassen

---



expanded class *INTEGER* ...

(intern: *INTEGER\_32*, *INTEGER\_64* etc.)

expanded class *BOOLEAN* ...

expanded class *CHARACTER* ...

expanded class *REAL* ...

(intern: *REAL\_32*, *REAL\_64* etc.)

*n: INTEGER*

Automatische Initialisierungsregeln:

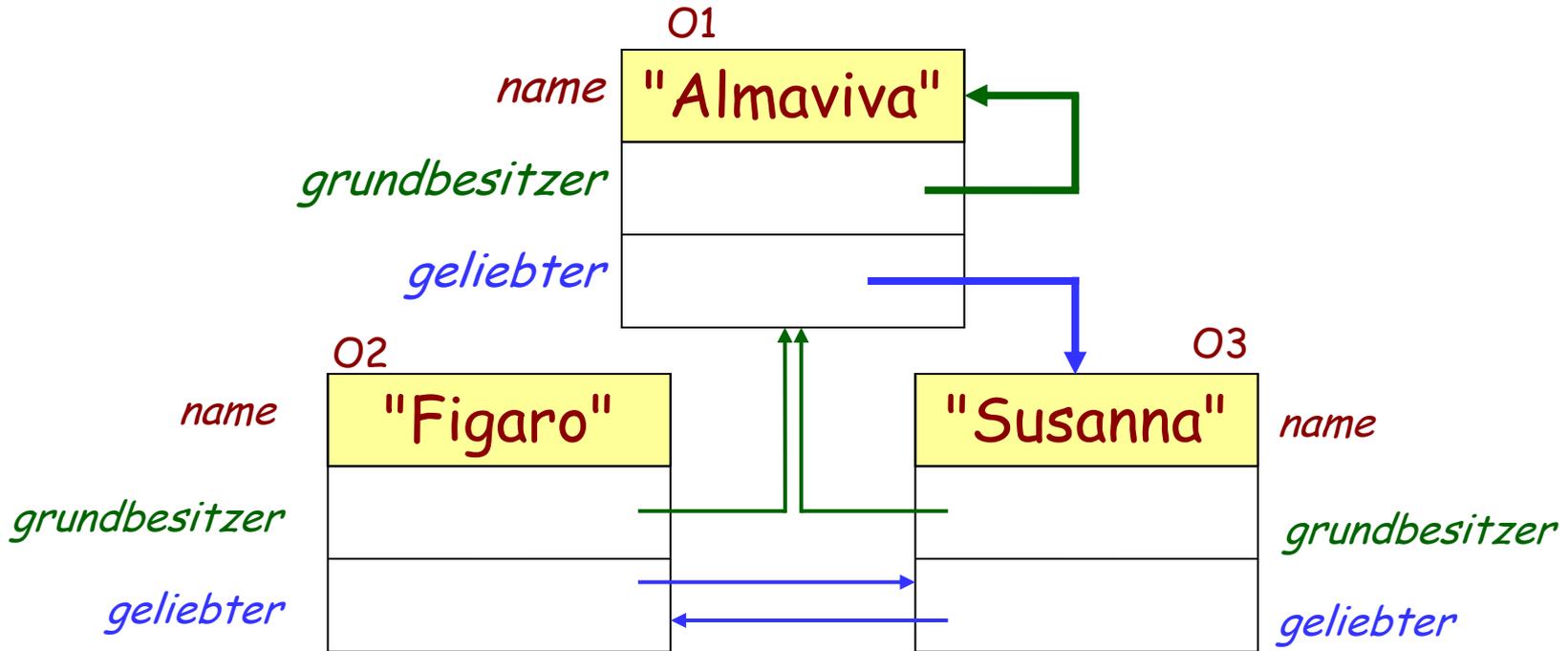
- 0 für Zahlen (ganze Zahlen, reelle Zahlen)
- "Null"-Zeichen für Zeichen
- **False** für Boole'sche Werte
- **Void** für Referenzen

Diese Regeln gelten für:

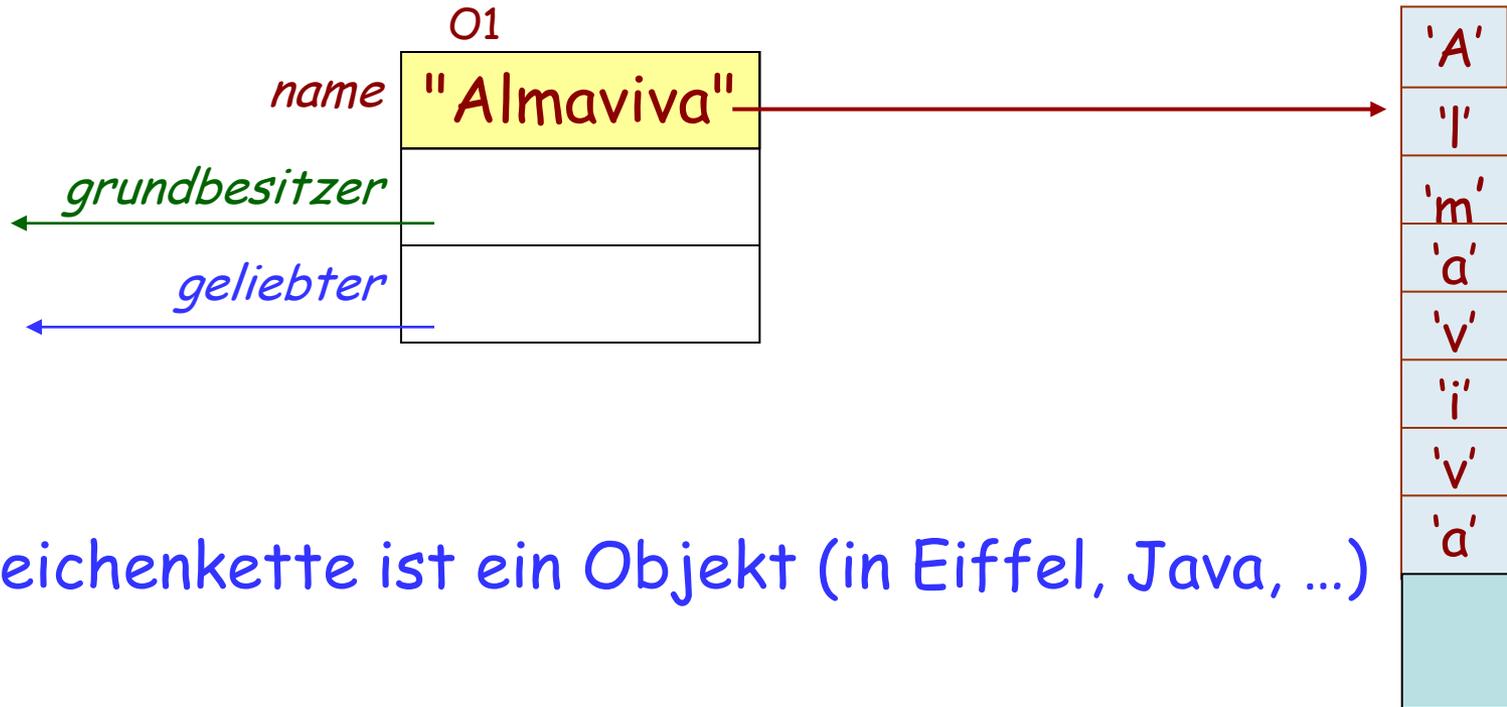
- Felder (von Klassenattributen), bei der **Objekterzeugung**
- Lokale Variablen, beim **Start der Routinenausführung**

(gilt auch für **Result**)

# Referenzen können Zyklen bilden



# Und was ist mit Zeichenketten (strings)?



Eine Zeichenkette ist ein Objekt (in Eiffel, Java, ...)

Das Feld *name* ist ein Referenzfeld

# Felder widerspiegeln Attribute der Klasse



```
class  
  VECTOR  
feature -- Zugriff  
  x: REAL  
    -- Östliche Koordinate.  
  y: REAL  
    -- Nördliche Koordinate.
```

Ein Attribut

Noch ein Attribut

Attribute sind Features  
der Klasse.

```
end
```

# Feldern einen Wert zuweisen (*assign*)



```
class
  VECTOR
feature -- Zugriff
  x: REAL
    -- Östliche Koordinate.
  y: REAL
    -- Nördliche Koordinate.

feature -- Element-Veränderungen
  set (new_x, new_y: REAL)
    do
      -- Setze Koordinaten auf [new_x, new_y].
    ...
  ensure
    x_gesetzt: x = new_x
    y_gesetzt: y = new_y
  end
end
```

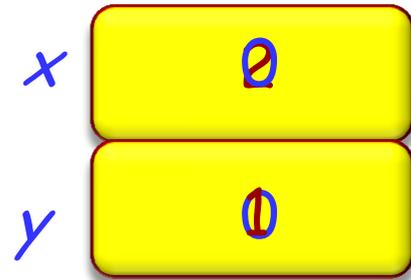
# Feldern einen Wert zuweisen



```
class
  VECTOR
feature -- Zugriff
  x: REAL
    -- Östliche Koordinate.
  y: REAL
    -- Nordliche Koordinate.

feature -- Element-Veränderungen
  set (new_x, new_y: REAL)
    -- Setze Koordinaten
    -- auf [new_x, new_y].
  do
    x := new_x
    y := new_y
  ensure
    x_gesetzt: x = new_x
    y_gesetzt: y = new_y
  end
end
```

Eine Instanz  
von *VECTOR*



Ausführung von *set*(2, 1)  
auf diese Instanz

# Was Sie tun können



```
class STATION feature
```

```
  name: STRING
```

```
    -- Name.
```

```
  position: VECTOR
```

```
    -- Position in Bezug auf das Stadtzentrum.
```

```
  set_position(new_x, new_y: REAL)
```

```
    -- Position setzen.
```

```
  do
```

```
    position.set(new_x, new_y)
```

```
  end
```

```
end
```

Besser:

-- Position auf Abzisse *new\_x*,  
-- Ordinate *new\_y* setzen.

# Qualifizierte und unqualifizierte Featureaufrufe



In Klasse *VECTOR*:

```
set (new_x, new_y: REAL)
do
...
end
```

In Klasse *VECTOR*:

```
move (dx, dy: REAL)
-- Um dx horizontal und dy vertikal verschieben.
```

```
do
  set (x+ dx, y+ dy)
ensure
  ... [Bitte ausfüllen!] ...
end
```

Unqualifizierter  
Aufruf

In einer anderen Klasse, z.B. *STATION*

```
position: VECTOR
set_position (new_x, new_y: REAL)
do
  position.set (new_x, new_y)
end
```

Qualifizierter  
Aufruf

# Das aktuelle Objekt (current object)

---



Zu jeder Zeit während einer Ausführung gibt es ein **aktuelles Objekt**, auf welchem das aktuelle Feature aufgerufen wird.

Zu Beginn: Das Wurzelobjekt. Danach:

- Ein **unqualifizierter Aufruf** wie z.B. *set(u, v)* wird auf das aktuelle Objekt angewendet.
- Ein **qualifizierter Aufruf** wie z.B. *x.set(u, v)* bedingt, dass das an *x* gebundene Objekt zum aktuellen Objekt wird. Nach dem Aufruf wird das vorherige Objekt wieder zum aktuellen Objekt.

Um auf das aktuelle Objekt zuzugreifen: Benutzen Sie **Current**

# Ausführung eines Systems



*Wurzelobjekt*



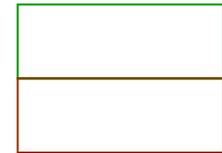
Wurzelprozedur



create *obj1.r1* *obj1*

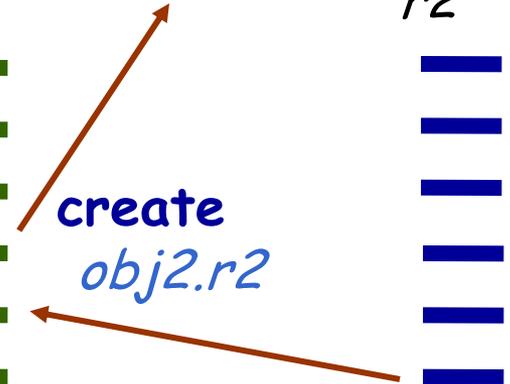


*obj2*



*r2*

create  
*obj2.r2*





Zu jeder Zeit während einer Ausführung gibt es ein **aktuelles Objekt**, auf welchem das aktuelle Feature aufgerufen wird

Zu Beginn ist dies das Wurzelobjekt

Während eines „qualifizierten“ Aufrufs  $x.f(a)$  ist das neue aktuelle Objekt dasjenige, das an  $x$  gebunden ist

Nach einem solchen Aufruf übernimmt das vorherige aktuelle Objekt wieder seine Rolle

**„Allgemeine Relativität“**

In *STATION*:

*position: VECTOR*

*set\_position(new\_x, new\_y: REAL)*

do

*position.set(new\_x, new\_y)*

end

# Die Kundenbeziehung (*client relation*)

---

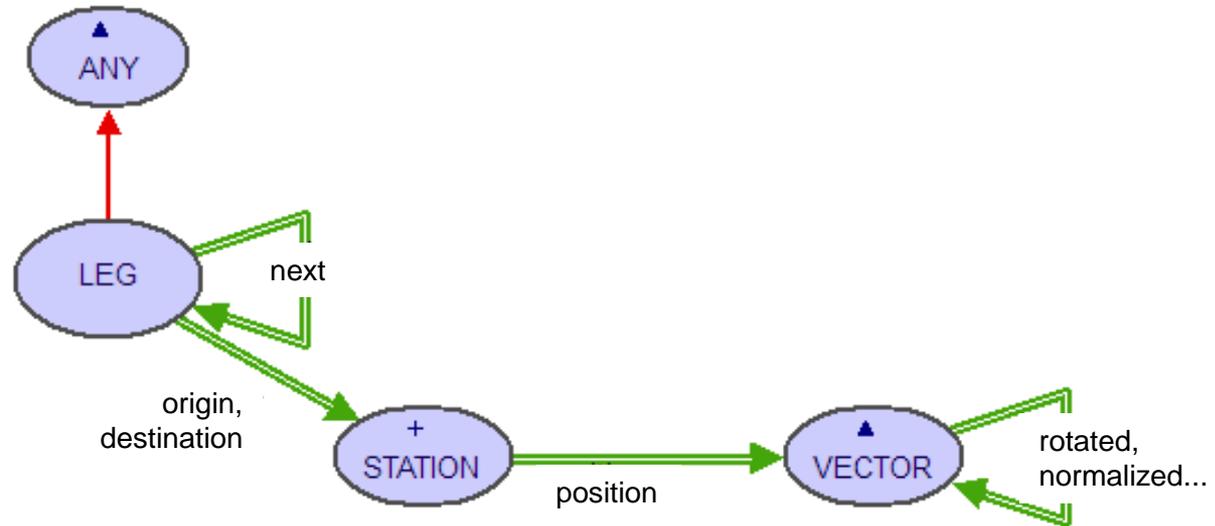


Da die Klasse *STATION* ein Feature

*position: VECTOR*

hat (und Aufrufe der Form *position.set(...)*), ist *STATION* ein Kunde der Klasse *VECTOR*

# Kunden und Vererbung - graphisch



Kunde



Vererbung





Eine Entität ist ein Name im Programm, der mögliche Laufzeitwerte bezeichnet

Manche Entitäten sind **konstant**

Andere sind **variabel**:

- Attribute
- Lokale Variablen



*ziel := quelle*

*quelle* ist ein **Ausdruck**, z.B.:

- Das Ergebnis einer Abfrage:
  - *position.x*
  - *i\_th(5)*
- Arithmetische oder Boole'sche Ausdrücke:
  - $a + (b * c)$
  - $(a < b) \text{ and } (c = d)$

*ziel* ist eine **Variable**. Die Möglichkeiten sind:

- Ein **Attribut**
- **Result** in einer Funktion (noch nicht gesehen bisher)
- Eine "**lokale Variable**" einer Routine (noch nicht gesehen)

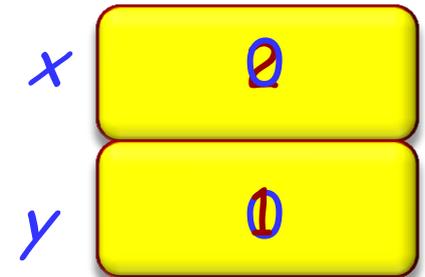
# Zuweisungen (Erinnerung)



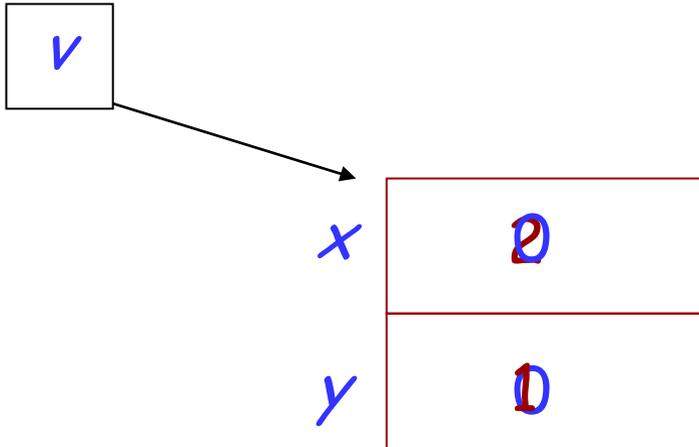
```
class
  VECTOR
feature -- Zugriff
  x: REAL
    -- Östliche Koordinate.
  y: REAL
    -- Nördliche Koordinate.

feature -- Element-Veränderungen
  set (new_x, new_y: REAL)
    -- Setze Koordinaten auf [new_x, new_y].
  do
    x := new_x
    y := new_y
  ensure
    x_gesetzt: x = new_x
    y_gesetzt: y = new_y
  end
end
```

Eine Instanz von *VECTOR*  
Ausführung von *set* (2, 1)  
auf diese Instanz



Eine Zuweisung ist eine Instruktion, die einen Wert durch einen anderen ersetzt.



`v.set(2, 1)`

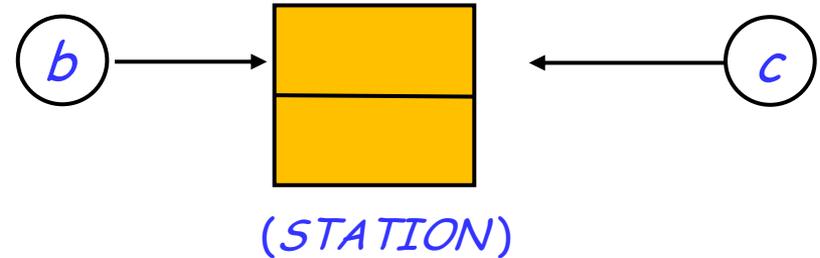
```
set(new_x, new_y: REAL)
do
  x := new_x
  y := new_y
end
```

# Zwei Arten von Typen



**Referenztypen:** Der Wert jeder Entität ist eine Referenz

*b: STATION*

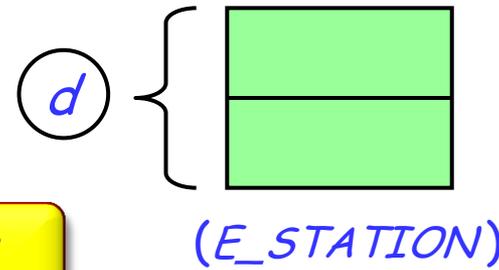


Zuweisung: kopiert die Referenz

*c := b*

**Expandierte Typen:** Der Wert jeder Entität ist ein Objekt

*d: E\_STATION*



Zuweisung: kopiert das Objekt

# Zuweisungen nicht mit Gleichheit verwechseln!



$x := y$

Instruktion

if  $x = y$  then...

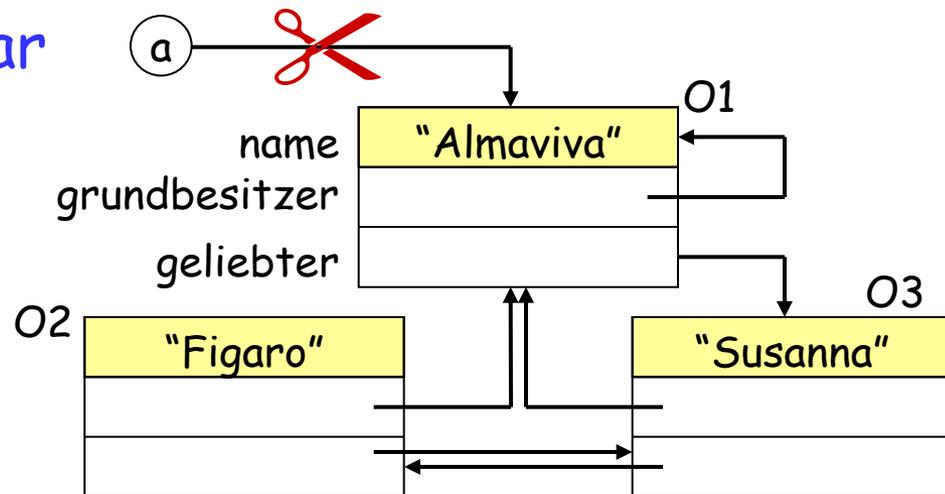
Ausdruck

if  $x = \text{Current}$  then...

# Was mit unerreichbaren Objekten tun?



Referenzzuweisungen können manche Objekte unerreichbar machen



Zwei mögliche Ansätze

- Manuelles "free" (C++, Pascal)
- Automatische Speicherbereinigung (garbage collection, d.h. Müllabfuhr) (Eiffel, Java, C#, .NET)



Newsgroup-Eintrag von Ian Stephenson, 1993 (Zitat aus *Object-Oriented Software Construction*):

*I say a big NO! Leaving an unreferenced object around is BAD PROGRAMMING. Object pointers ARE like ordinary pointers — if you allocate an object you should be responsible for it, and free it when its finished with. (Didn't your mother always tell you to put your toys away when you'd finished with them?)*

# Argumente für automatische Bereinigung

---



Manuelle Bereinigung ist eine Gefahr für die Zuverlässigkeit:

- Falsche "frees" sind Bugs, die nur sehr schwierig zu finden und zu beheben sind.

Manuelle Bereinigung ist mühsam.

Moderne (automatische) Speicherbereiniger haben eine akzeptable Performance.

Speicherbereinigung ist einstellbar: an/abschalten, parametrisieren...

# Eigenschaften einer Speicherbereinigung

---



Konsistenz :

*Nur* unerreichbare Objekte werden bereinigt

Vollständigkeit :

*Alle* unerreichbaren Objekte werden bereinigt

Konsistenz (*Soundness*) ist eine absolute Anforderung.  
Lieber keine Speicherbereinigung als eine unsichere  
Speicherbereinigung

Aber: sichere automatische Speicherbereinigung ist  
schwierig für C-basierte Sprachen

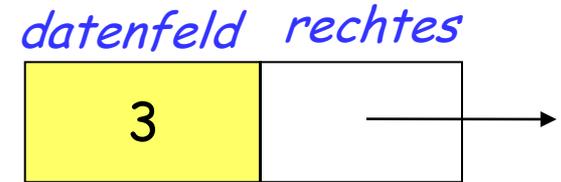
# Effekt einer Zuweisung



Referenztypen: Referenzzuweisung

Expandierte Typen: Kopie des Wertes

```
class LINKABLE
feature
  item: INTEGER
  right: LINKABLE
  set (n : INTEGER ; r : LINKABLE)
    -- Beide Felder setzen.
  do
    item := n
    right := r
  end
end
```



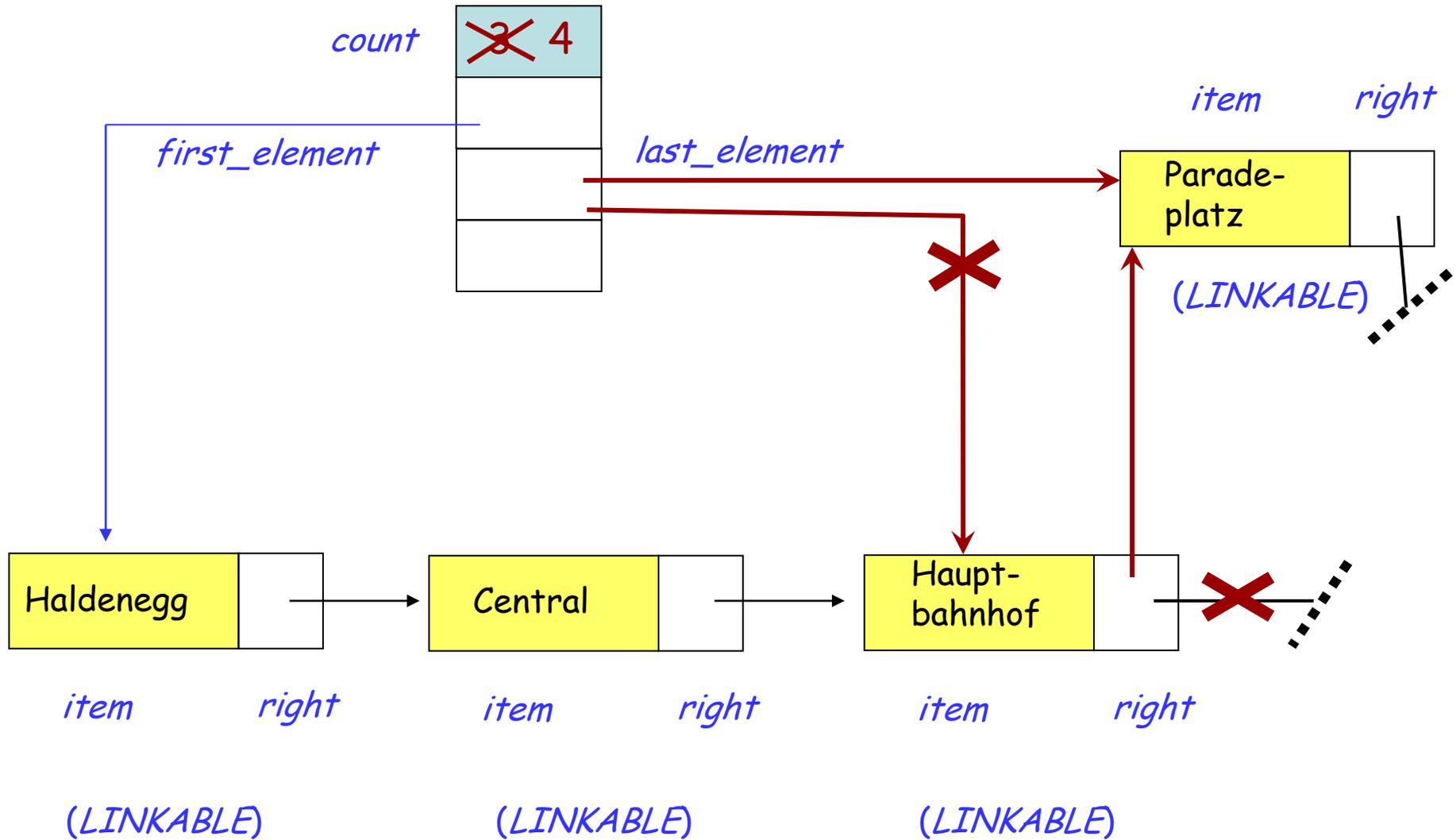
```
l: LINKABLE
...
create l
...
l.set(25, Void)
```



```
class LINE feature
  number: INTEGER
  color: COLOR
  city: CITY

  set_all (n: INTEGER; co: COLOR; ci: CITY)
    do
      number := n
      color := co
      city := ci
    end
end
```

# Eine verkettete Liste von Strings: Einfügen am Ende



# Ein Element am Ende einfügen



```
extend(v: STRING)
    -- v am Ende hinzufügen.
    -- Den Cursor nicht verschieben.
local
    p: LINKABLE[STRING]
do
    create p.make(v)
    if is_empty then
        first_element := p
    else
        last_element.put_right(p)
    end
    last_element := p
    count := count + 1
end
```

# LINKABLE - Zellen



```
class LINKABLE feature
```

```
  item: STRING
```

```
    -- Wert dieser Zelle.
```



*item*

*right*

```
  right: LINKABLE
```

```
    -- Zelle, welche an diese Zelle angehängt ist  
    -- (falls vorhanden).
```

```
  put_right(other: like Current)
```

```
    -- Setzt other rechts neben die aktuelle Zelle.
```

```
  do
```

```
    right := other
```

```
  ensure
```

```
    angehaengt: right = other
```

```
  end
```

```
end
```

# Lokale Variablen (in Routinen)



Deklarieren Sie einfach zu Beginn einer Routine (in der **local** Klausel)

```
r(...)
    require      -- Kopfkomentar
    local      ...
                x: REAL
                m: STATION
    do
                ... x und m sind hier benutzbar ...
    ensure
                ... x und m sind hier nicht mehr benutzbar ...
    end
```

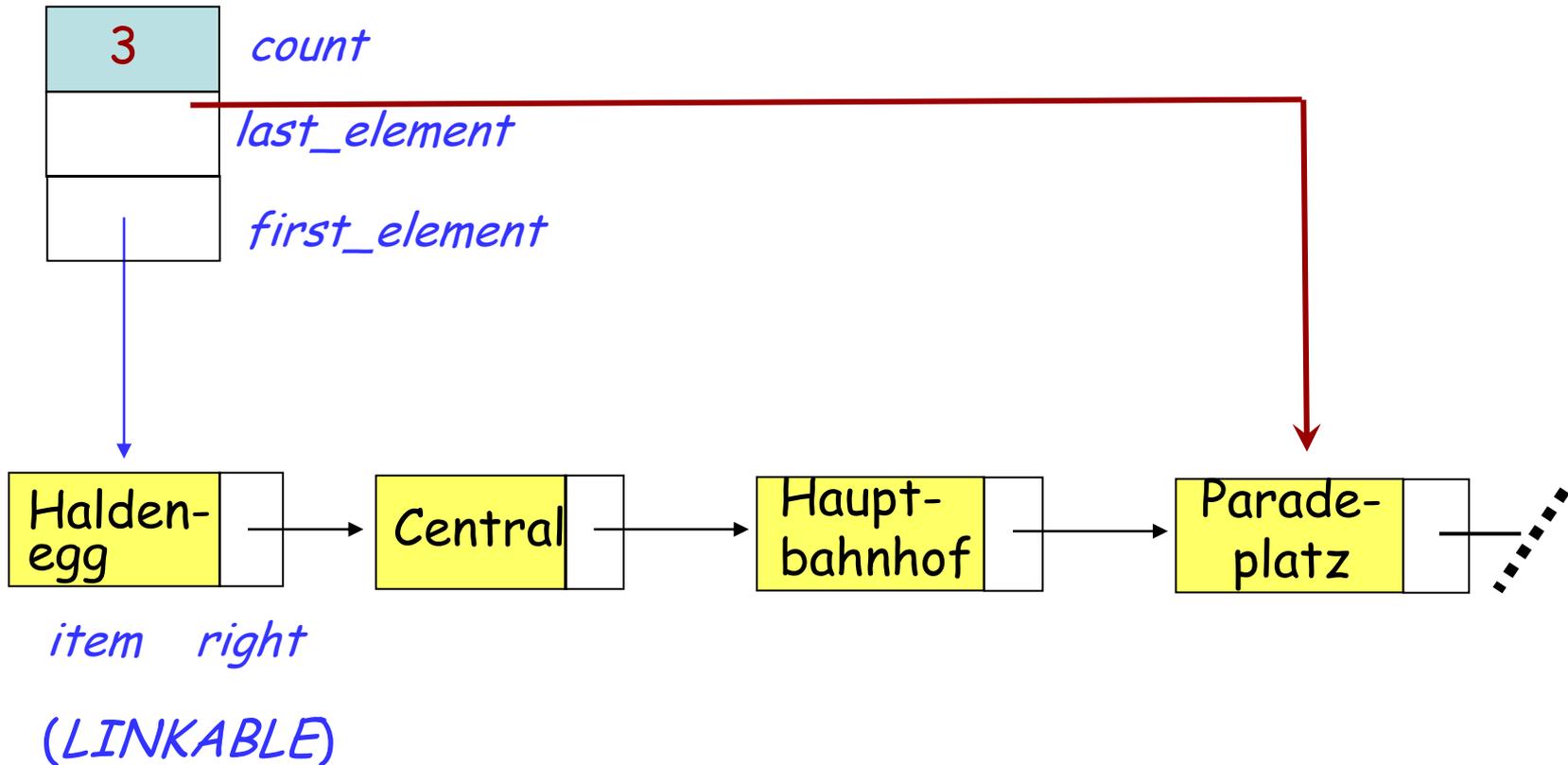
**Result** ist (für Funktionen) auch eine lokale Variable.

# Übung (beinhaltet Schleifen)



Kehren Sie eine Liste um!

(LINKED\_LIST)



# Leseaufgaben für die nächsten 2 Wochen

---



Control structures (Kapitel 7)



- Das aktuelle Objekt
- Expandierte Typen und Referenztypen
- Zuweisung:
  - Für Referenzen
  - Für expandierte Werte
- Verkettete Datenstrukturen
- Einen kurzen Blick auf bedingte Anweisungen