



Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lecture 10: Das dynamische
Modell und mehr zu Referenzen

Ziel dieser Vorlesung

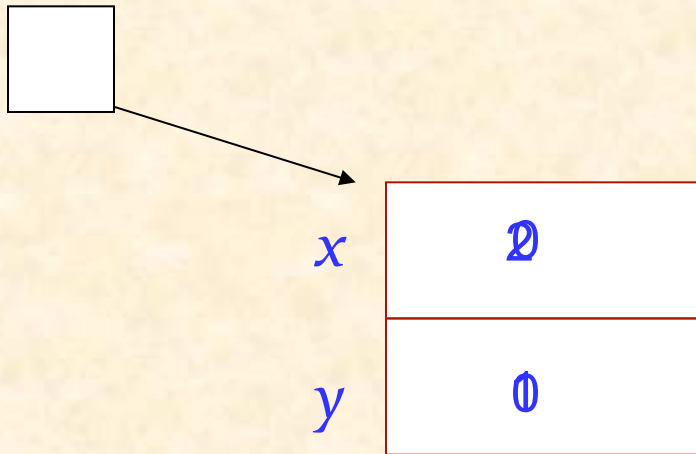


Ein paar neue Konzepte und insbesondere ein besseres Verständnis des recht schwierigen Themas der **Referenzen**

Anmerkungen zur Speicherbereinigung und zugehörigen Konzepten



Ersetzt einen Wert durch einen anderen



p.set_coordinates (2, 1)

Feldern einen Wert zuweisen (in einer Routine)

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

Effekt einer Zuweisung

Referenztypen: Referenzzuweisung

Expandierte Typen: Kopie des Wertes

Siehe *LINKABLE* in EiffelBase

```
class LINKED_CELL feature
```

```
  item : INTEGER
```

```
  right : LINKED_CELL
```

```
  set_fields (n : INTEGER ; r : LINKED_CELL)
```

```
    -- Beide Felder neu setzen.
```

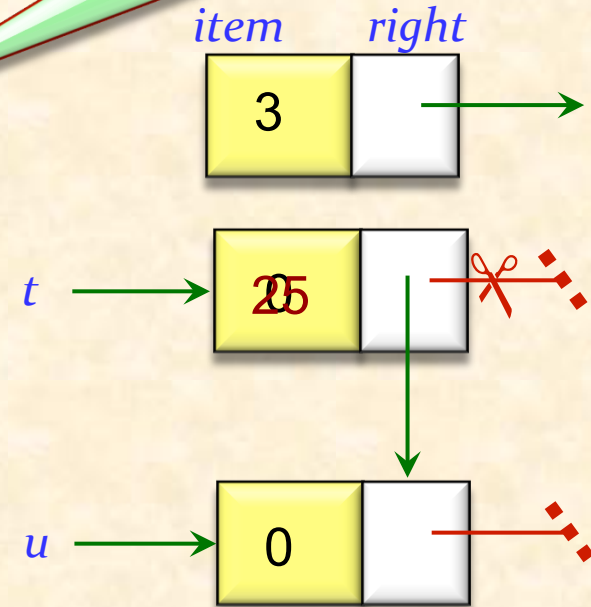
```
do
```

```
  item := item := n
```

```
  right := right := r
```

```
end
```

```
end
```

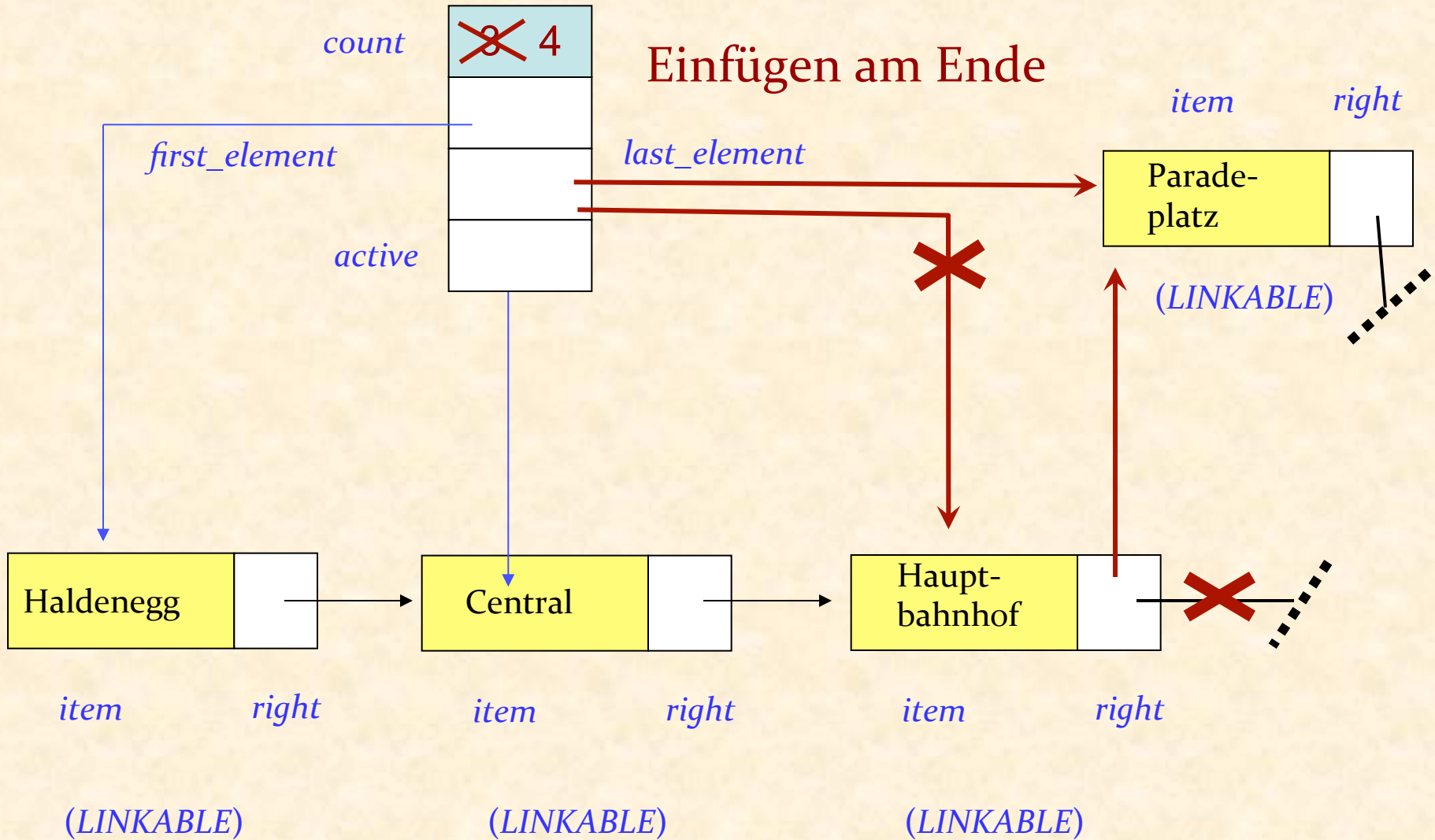


```
t, u : LINKED_CELL
```

```
create t ; create u
```

```
t.set_fields (25, u)
```

Eine verkettete Liste von Strings:



Ein Element am Ende einfügen



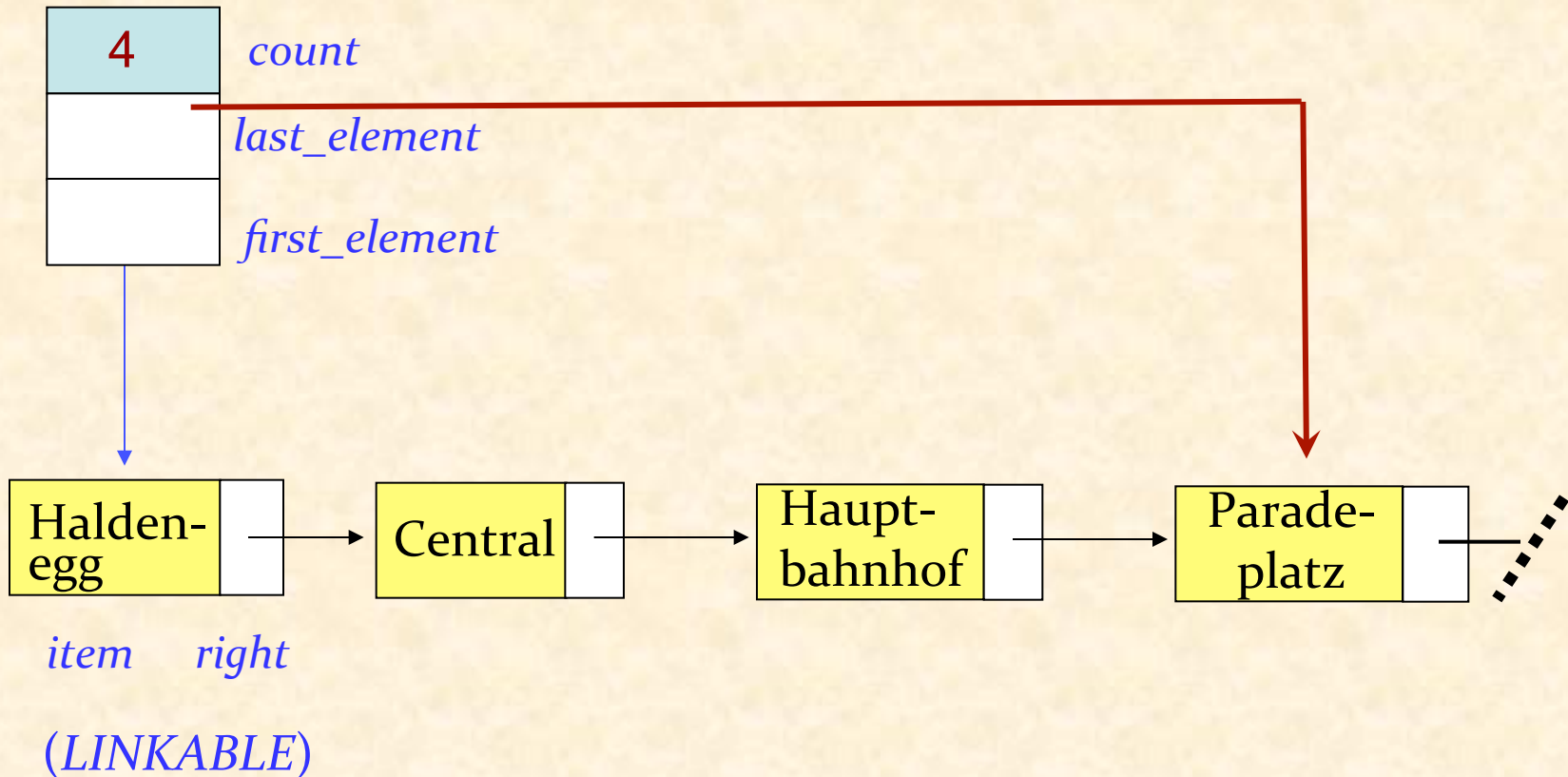
```
extend (v : STRING)  
    -- Füge v am Ende hinzu.  
    -- Cursor nicht verschieben.  
local  
    p : LINKABLE [STRING]  
do  
    create p.make (v)  
    if is_empty then  
        first_element := p  
        active := p  
    else  
        last_element.put_right (p)  
        if after then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

Übung (beinhaltet Schleifen)

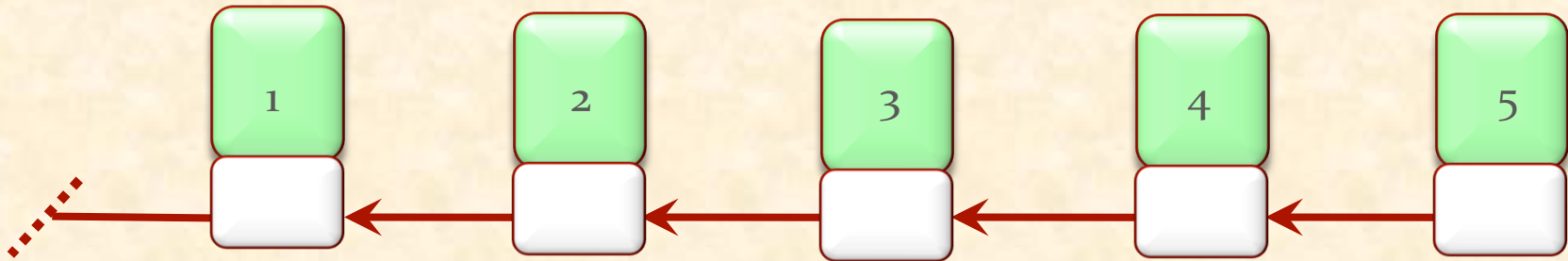
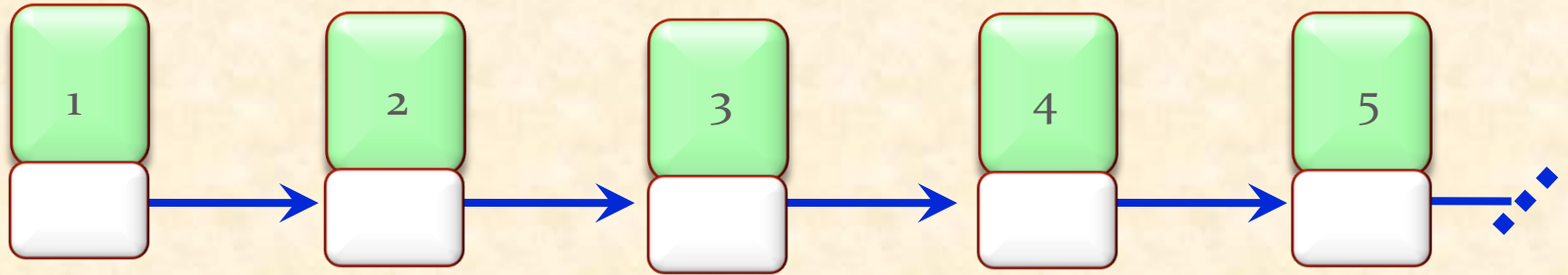


Kehren Sie eine Liste um!

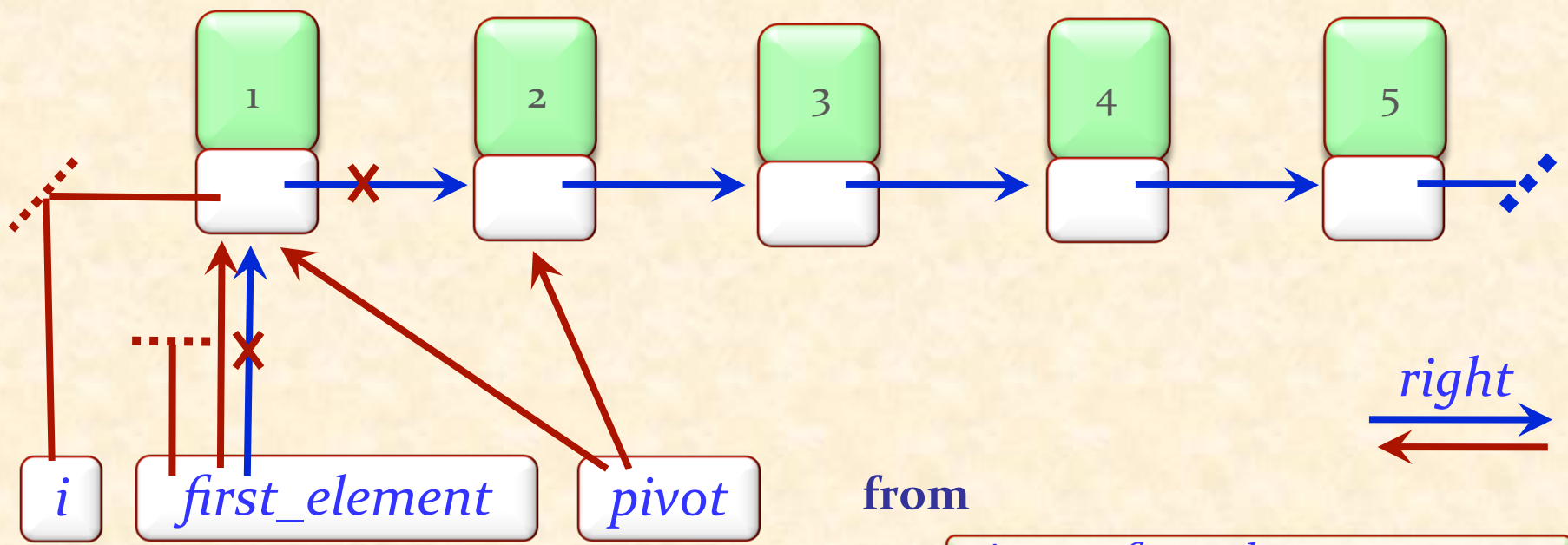
(LINKED_LIST)



Eine Liste umkehren



Eine Liste umkehren



from

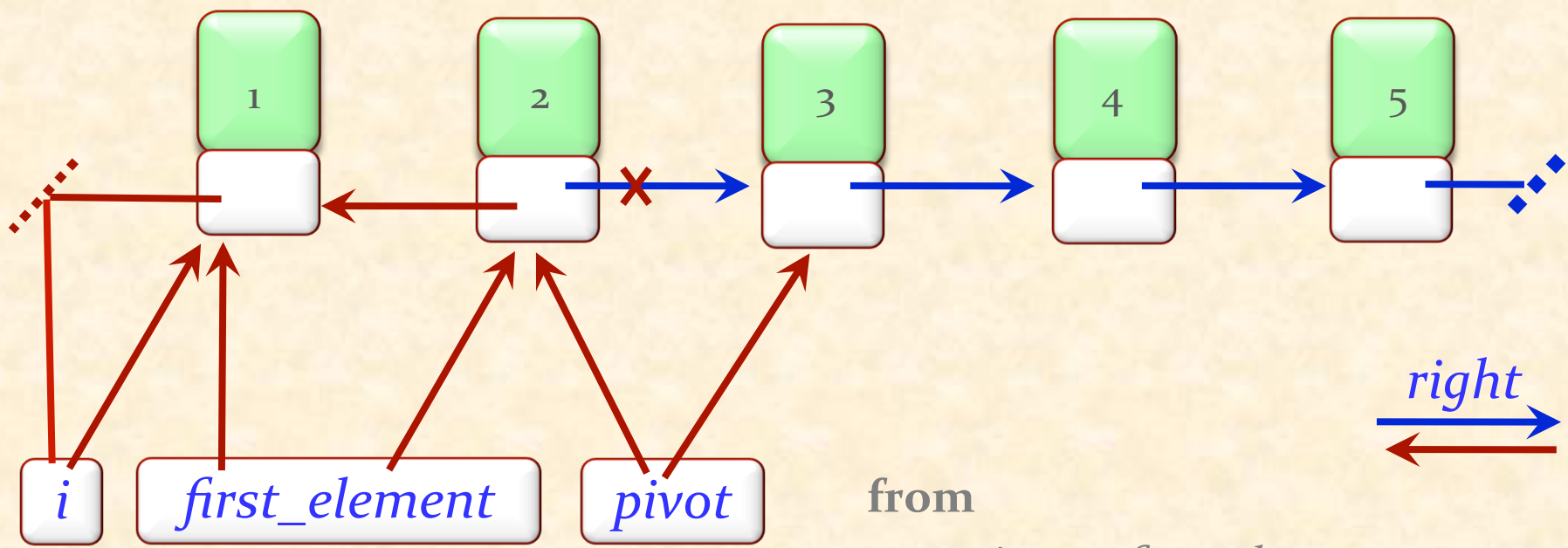
```
pivot := first_element  
first_element := Void
```

until *pivot* = Void loop

```
i := first_element  
first_element := pivot  
pivot := pivot.right  
first_element.put_right(i)
```

end

Eine Liste umkehren



from

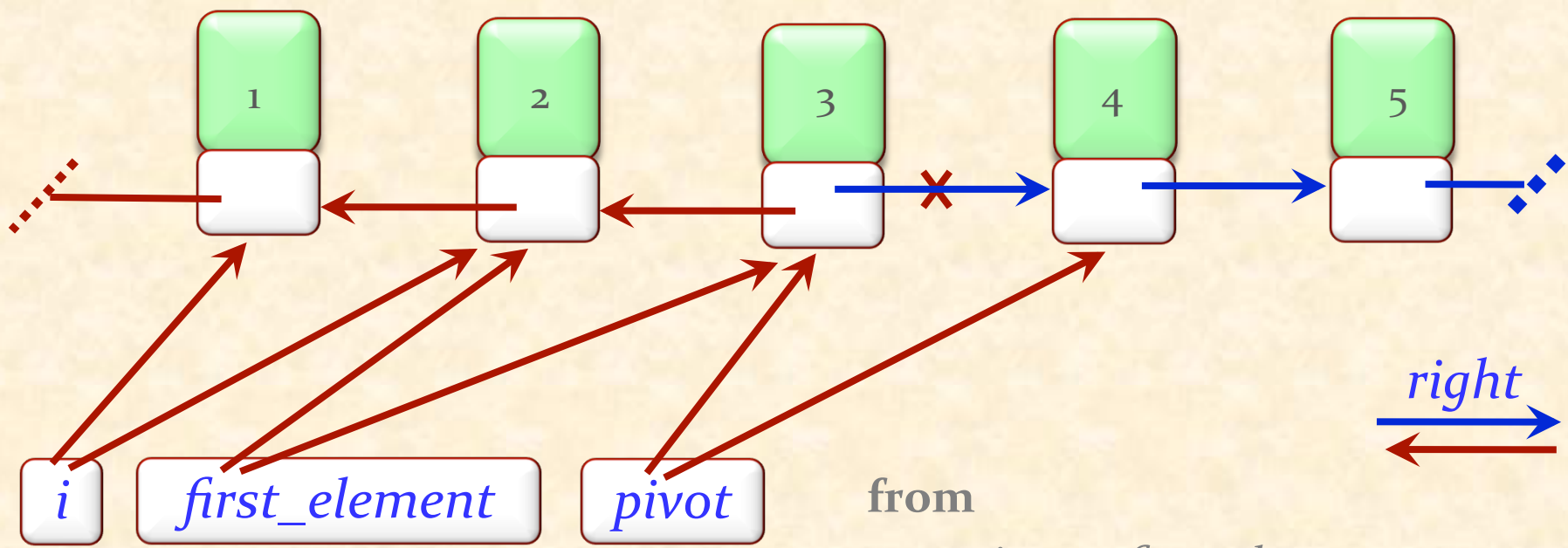
```
pivot := first_element  
first_element := Void
```

until *pivot = Void* **loop**

- i := first_element*
- first_element := pivot*
- pivot := pivot.right*
- first_element.put_right (i)*

end

Eine Liste umkehren



from

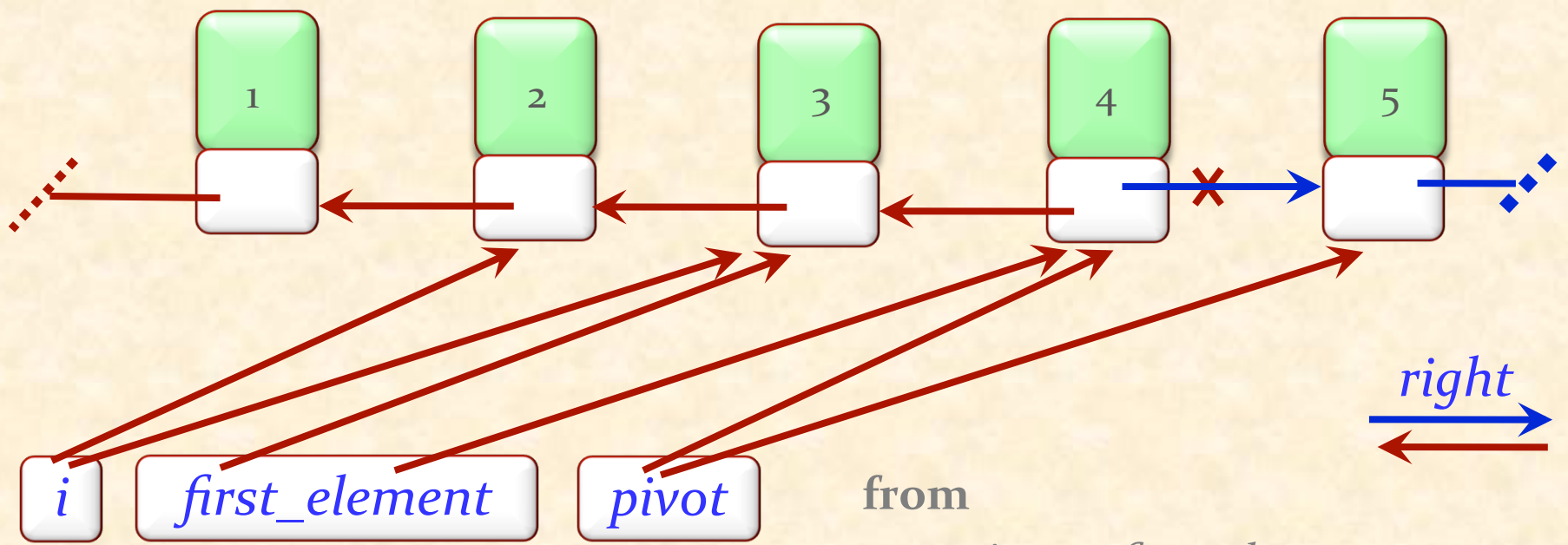
```
pivot := first_element  
first_element := Void
```

until *pivot* = Void loop

- i* := *first_element*
- first_element* := *pivot*
- pivot* := *pivot.right*
- first_element.put_right* (*i*)

end

Eine Liste umkehren



from

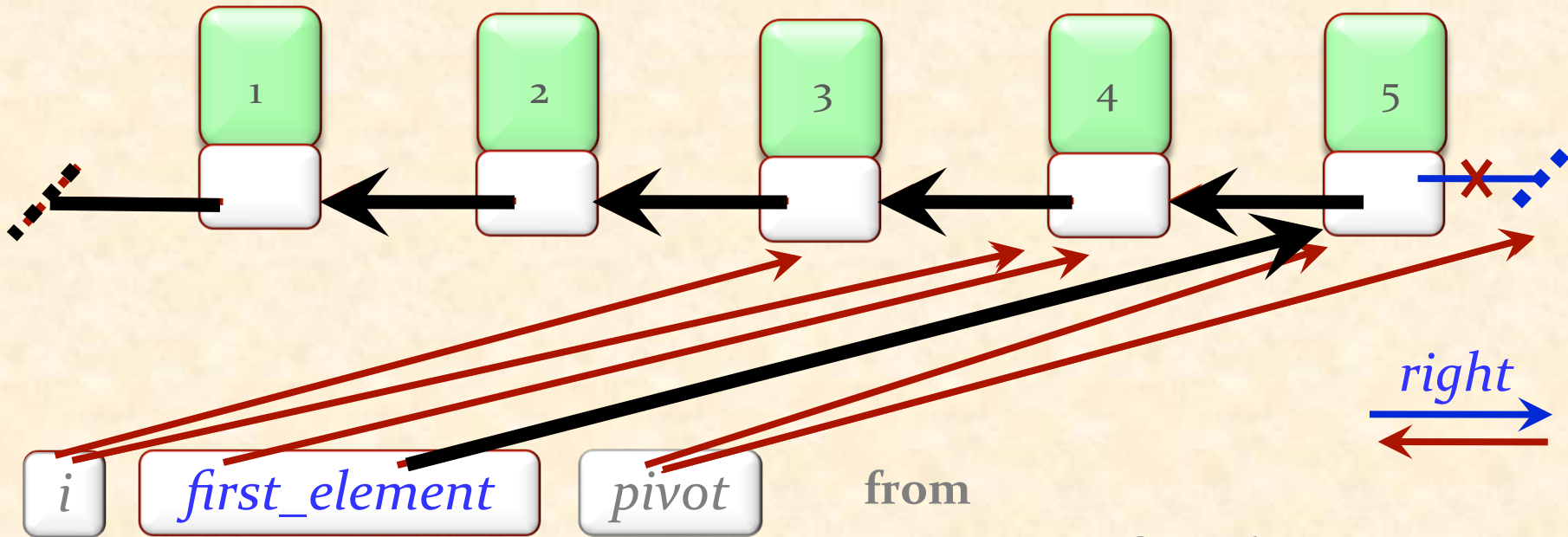
```
pivot := first_element  
first_element := Void
```

until *pivot* = Void loop

- i* := *first_element*
- first_element* := *pivot*
- pivot* := *pivot.right*
- first_element.put_right* (*i*)

end

Eine Liste umkehren



from

```
pivot := first_element  
first_element := Void
```

until *pivot* = Void loop

```
i := first_element
```

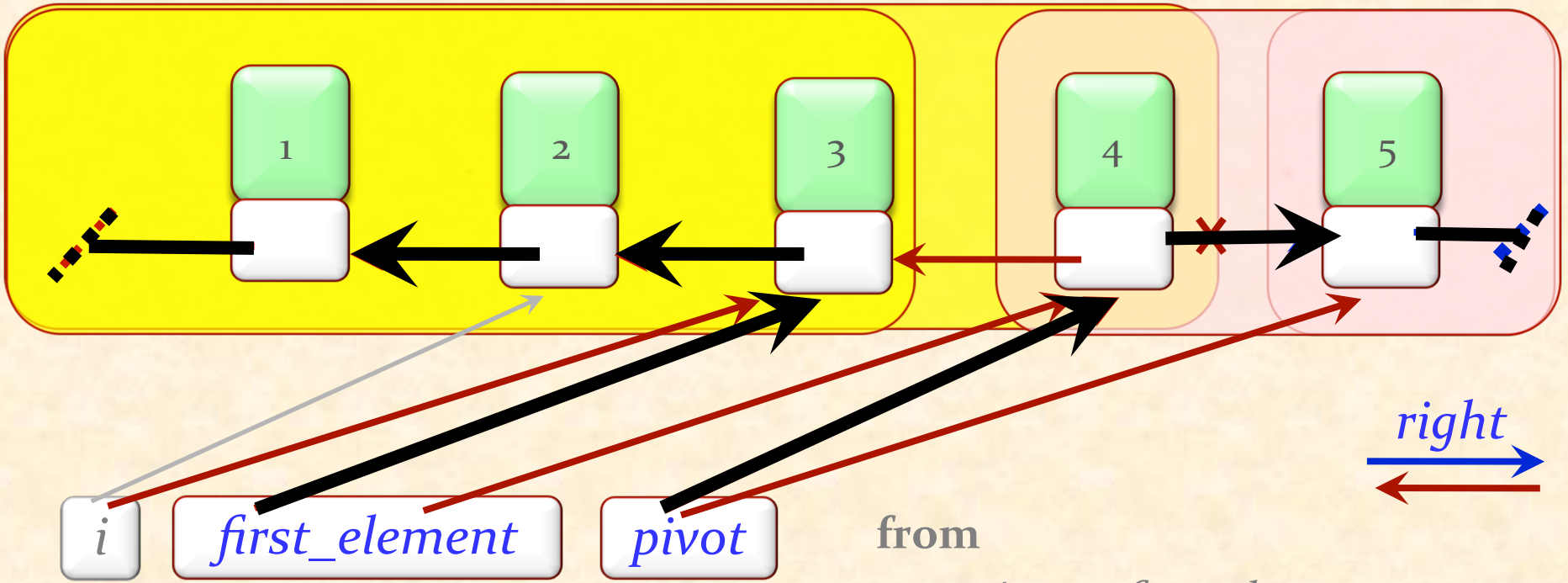
```
first_element := pivot
```

```
pivot := pivot.right
```

```
first_element.put_right (i)
```

end

Die Schleifeninvariante



Invariante:

- Von *first_element* nach *right*: anfängliche Elemente in umgekehrter Reihenfolge
- Von *pivot* aus: Rest der Elemente in ursprünglicher Reihenfolge.

from

```
pivot := first_element  
first_element := Void
```

until *pivot* = Void loop

```
i := first_element  
first_element := pivot  
pivot := pivot.right  
first_element.put_right(i)
```

end

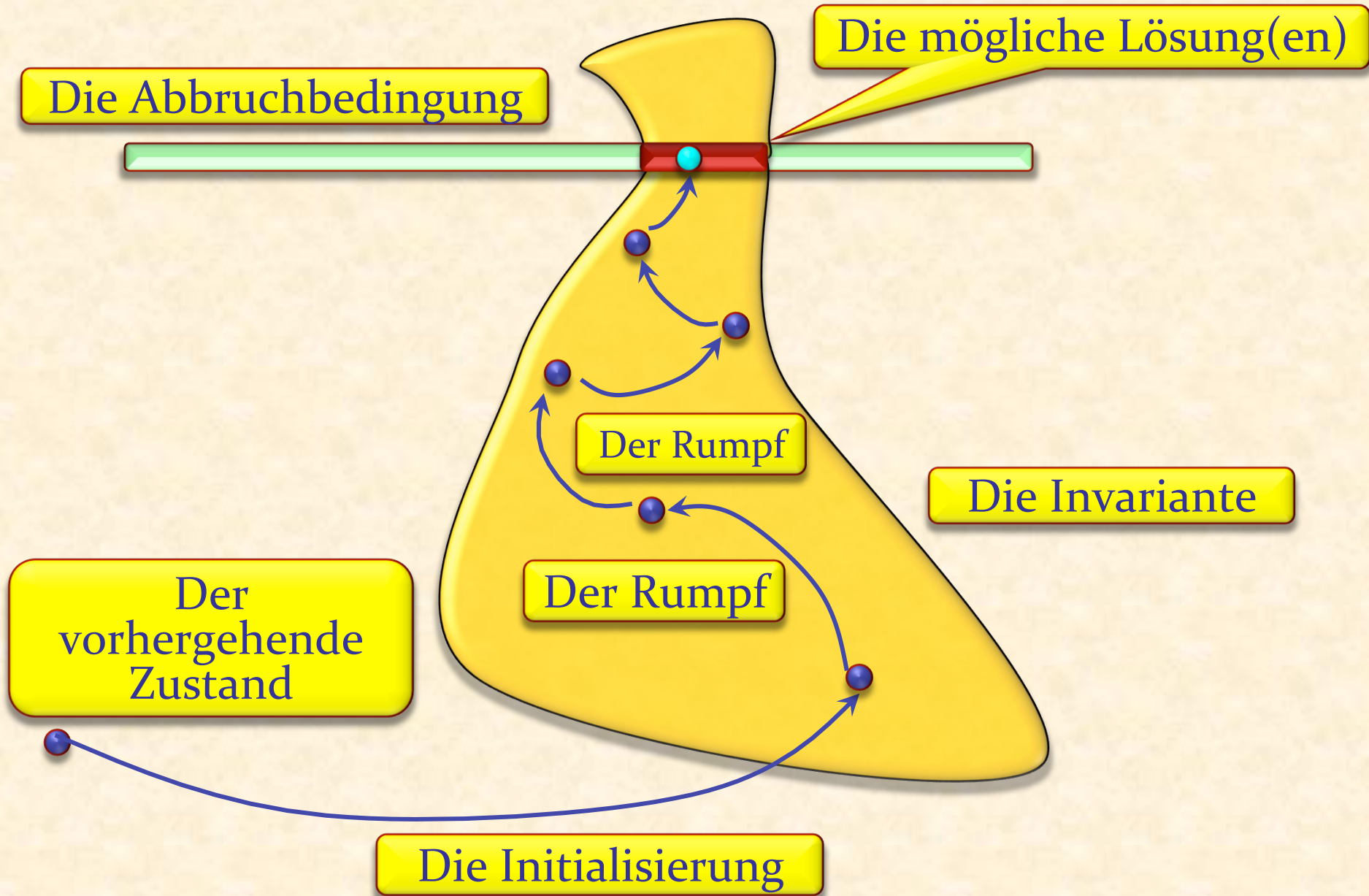


(Nicht zu verwechseln mit der Klasseninvariante)

Eine Eigenschaft, die:

- Nach der Initialisierung (**from**-Klausel) erfüllt ist
- Von jedem Schleifendurchlauf (**loop**-Klausel), bei der die Abbruchbedingung (**until**-Klausel) *nicht* erfüllt ist, eingehalten wird
- Wenn die Abbruchbedingung erfüllt ist, das gewünschte Ergebnis sicherstellt

Die Schleifeninvariante



Das Problem mit Referenzzuweisungen



Eine intuitive Argumentationsweise:

-- Hier ist *SOME_PROPERTY* für *a* erfüllt

“Wende *SOME_OPERATION* auf *b* an”

-- Hier gilt *SOME_PROPERTY* immer noch für *a*

Dies gilt für expandierte Werte, z.B. ganze Zahlen (INTEGER)

-- Hier ist *P* (*a*) erfüllt.

OP (*b*)

-- Hier ist *P* (*a*) immer noch erfüllt für *a*

Dynamische Mehrfachbenennung



a, b: LINKABLE [STRING]

create a....

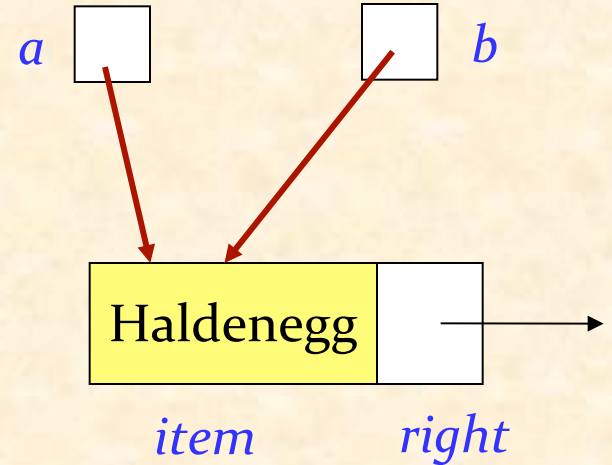
a.put ("Haldenegg")

b := a

-- Hier hat *a.item* den Wert "Haldenegg"

b.put ("Paradeplatz")

-- Hier hat *a.item* den Wert ?????



Andererseits...

- Ich habe gehört, dass die Cousine des Chefs weniger
- als 50'000 Franken pro Jahr verdient

“Erhöhen Sie Caroline’s jährliches Gehalt um 50 Rappen”

-- ?????

Metaphern:

- “Die schöne Tochter von Leda”
- “Menelas’ Gefährtin”
- “Geliebte von Paris”

= Helena von Troja



Referenzzuweisungen sind nützlich

Sie sind möglicherweise auch etwas schwieriger

Überlassen Sie diese so oft wie möglich den spezialisierten Bibliotheken von generellen Datenstrukturen.

Varianten von Zuweisungen und Kopieren

Referenzzuweisung (Typen von a und b sind Referenztypen):

$b := a$

Duplizieren eines Objektes (flach):

$c := a.tw\textit{in}$

Duplizieren eines Objektes (tief):

$d := a.deep_tw\textit{in}$

Weiterhin: flache Feld-um-Feld Kopie (Kein neues Objekt wird erzeugt):

$e.copy(a)$

Flaches und tiefes Klonen



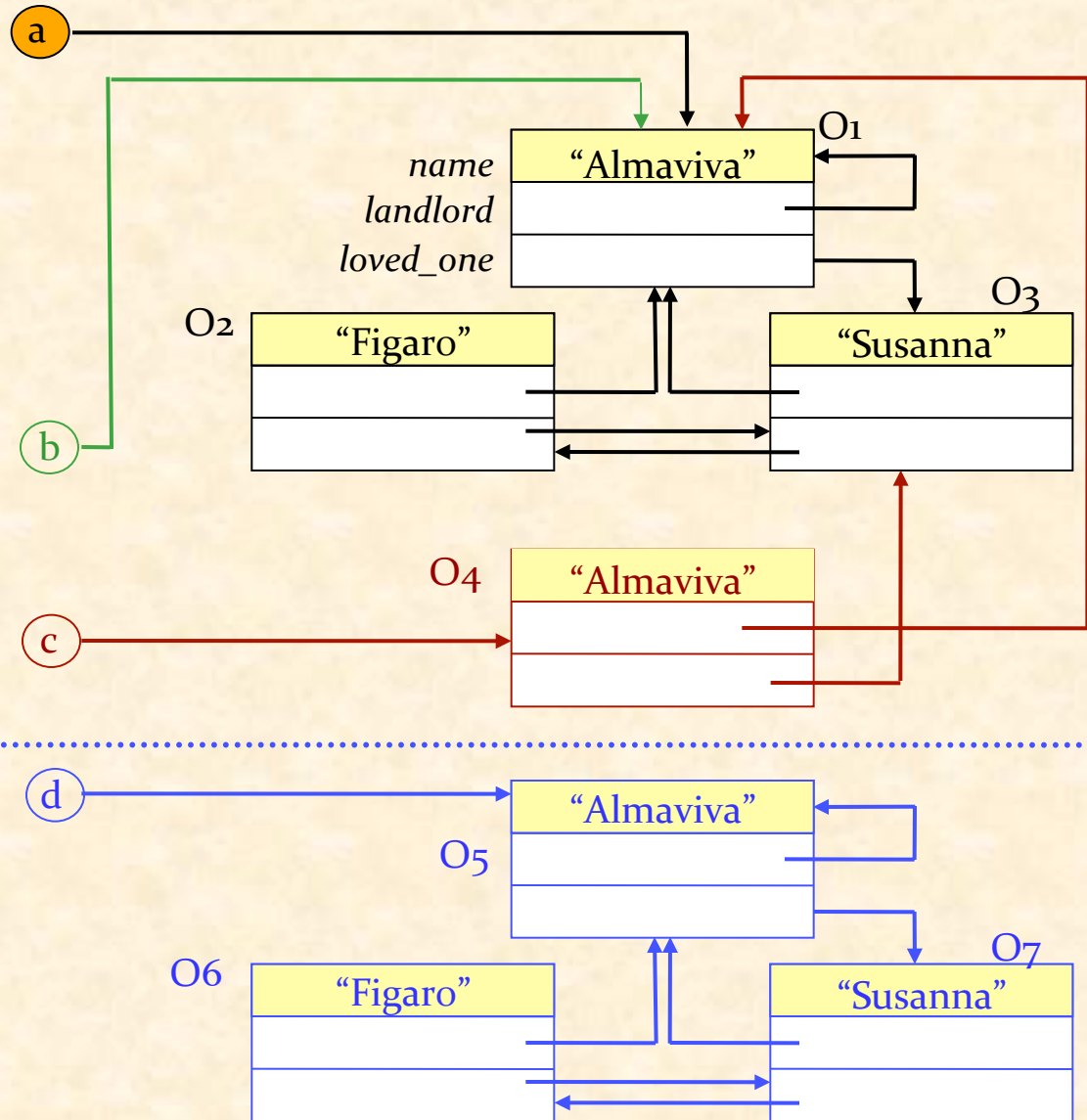
Anfangssituation:

Resultat von:

$b := a$

$c := a.\text{twin}$

$d := a.\text{deep_twin}$



Woher kommen diese Mechanismen?

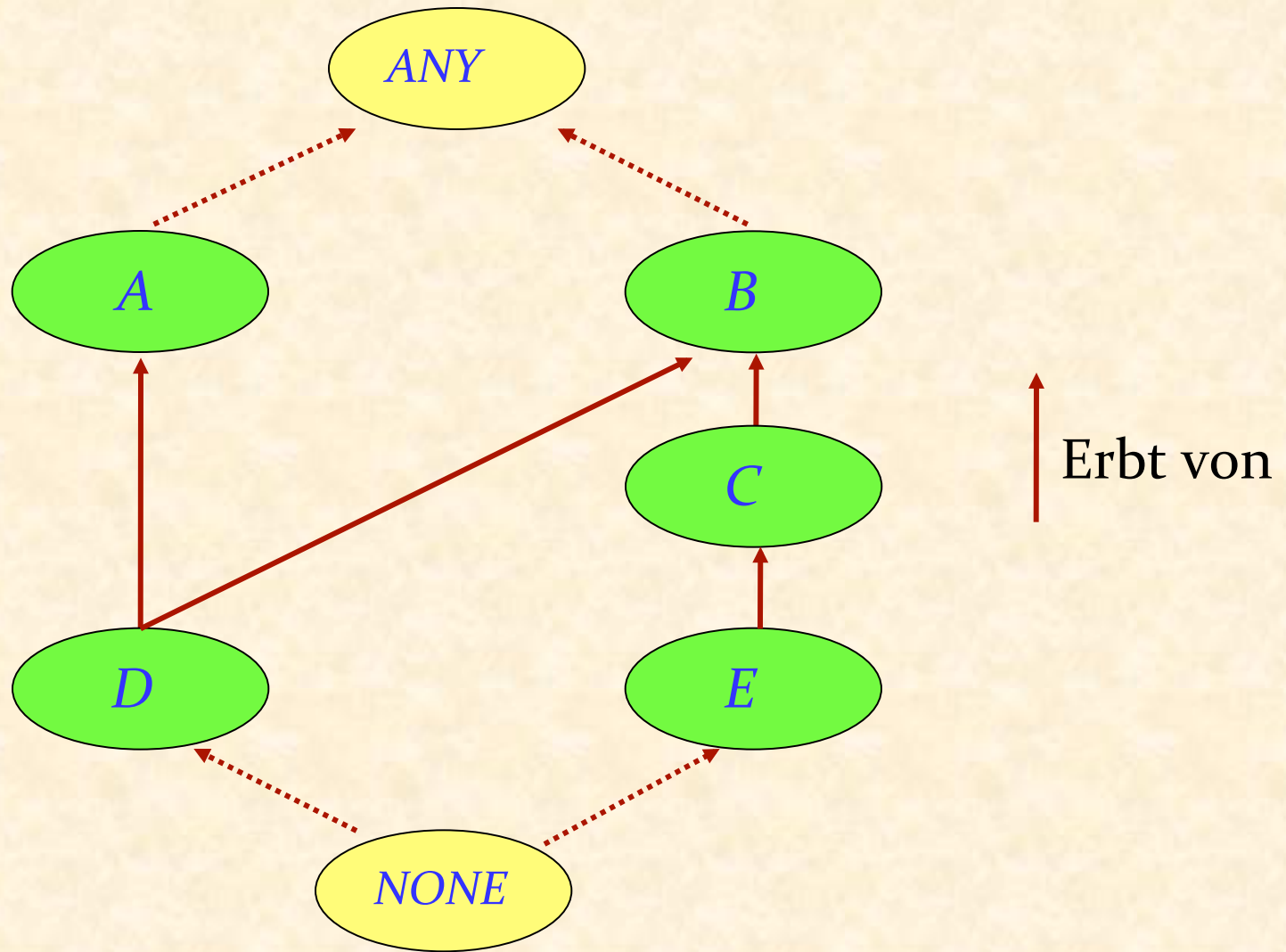


Die Klasse *ANY* in der Eiffel “Kernel-Bibliothek”

Jede Klasse, die nicht explizit von einer Anderen erbt, erbt implizit von *ANY*

Deshalb ist jede Klasse ein Nachkomme von *ANY*

Die Vererbungsstruktur vervollständigen



Ein verwandter Mechanismus: Persistenz



$a, b: X$

$a.store$ ("FN")

....

$b := retrieved$ ("FN")

Muss verbessert werden, siehe "Objekt-Test"

Die Speicherung erfolgt automatisch.

Persistente Objekte werden durch individuelle Schlüssel identifiziert.

Diese Features kommen aus der Bibliotheksklasse *STORABLE*.

Einen Typ erzwingen: Der Objekt-Test



Zu prüfender
Ausdruck

“Object-Test Local”

```
if attached {X} retrieved (“FN”) as r then
```

```
--Tu irgendwas mit r, welches garantiert nicht  
-- void und vom dynamischen Typ X ist.
```

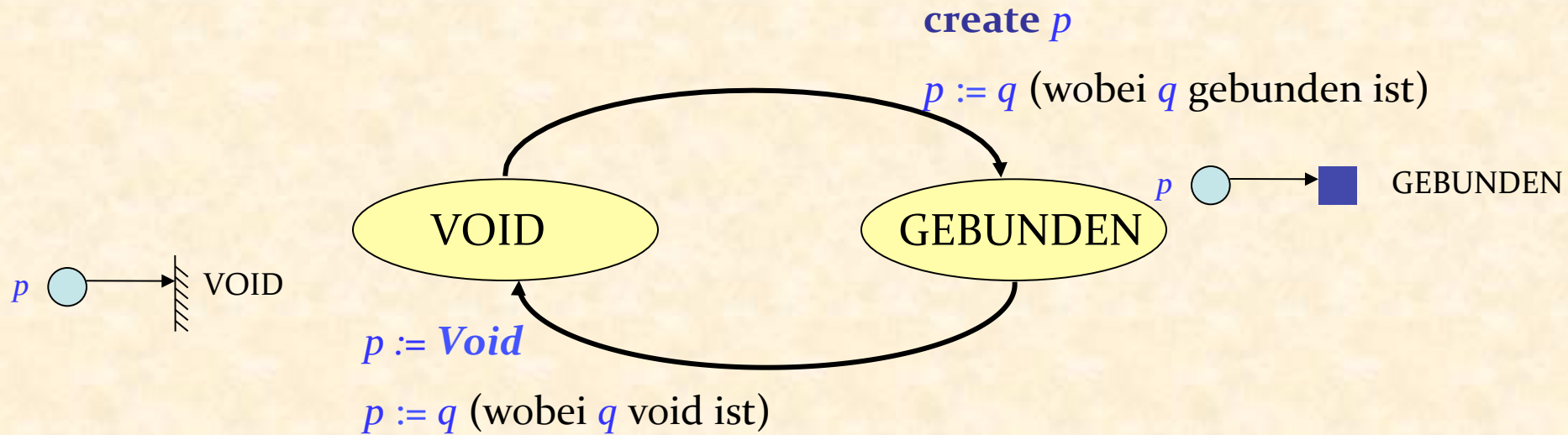
```
else
```

```
  print (“Too bad.”)
```

```
end
```

SCOPE der lokalen Variablen

Status einer Referenz:



Operationen auf Referenzen:

create p

$p := q$

$p := \text{Void}$

if $p = \text{Void}$ then ...

Die Objekt-Orientierte Form eines Aufrufs



some_target.some_feature (some_arguments)

Zum Beispiel:

Zurich_map.animate

Line10.append (Haldenbach)

x := a.plus (b) ????????



Bei

$$a - b$$

ist der $-$ Operator ein “**infix**“-Operator
(zwischen den Operanden geschrieben)

Bei

$$- b$$

ist der $-$ Operator ein “**präfix**“-Operator
(vor dem Operand geschrieben)



expanded class *INTEGER* feature

plus alias "+" (*other* : *INTEGER*): *INTEGER*

-- Summe mit *other*

do ... end

times alias "*" (*other* : *INTEGER*): *INTEGER*

-- Multiplikation mit *other*

do ... end

minus alias "-" : *INTEGER*

-- unäres Minus

do ... end

...

end

Aufrufe wie *i.plus* (*j*) können jetzt auch als *i + j* geschrieben werden



Chapters on

- Recursion(12)
- **Inheritance (16)**



- Mit Referenzen spielen: Umkehrung einer Liste
- Dynamische Mehrfachbenennung und die Schwierigkeiten von Zeigern und Referenzen
- Allgemeine Vererbungsstruktur
- Kopieren, Klonen und Speicheroperationen
- Persistenzhülle
- Infix- & Präfix-Operatoren