CLOS

Objektorientierte Programmierung in Lisp

Thomas Rühr, Lorenz Mösenlechner

awbs-is@in.tum.de

Überblick

- Klassen
- Instanzen
- Generische Funktionen
- Methodenkombination
- Vergleich zu Java

Klassen

- Allgemeine Klassendefinition:

```
(defclass NAME (SUPERCLASSES)
   (ATTRIBUTES))
```

- Attribute können so angegeben werden:
  - *NAME* (ohne Klammer, ohne alles)
  - *(NAME PROPERTIES)*
- einfaches Beispiel:

```
(defclass rectangle ()
   (height width))
```

Properties (1)

Als „Properties“ können angegeben werden:

- :accessor (Zugriffsmethode)
- :initarg (Initialisierungsname)
- :initform (Initialisierungswert)
- :allocation
  - Attribut, auf das alle Instanzen zugreifen *:class*
  - Standardmäßig *:instance*

Properties (2)

Beispiel:

```
(defclass circle ()
   ((radius :accessor circle-radius
            :initarg :radius
            :initform 1)
    (center :accessor circle-center
            :initarg :center
            :initform (cons 0 0))))
```

Instanzen

- Instanz erschaffen:

```
(make-instance CLASSNAME INITVALS)
```

  - *:initarg* angegeben $\Rightarrow$ entsprechender Wert
  - sonst:
    - *:initform* vorhanden $\Rightarrow$ entsprechender Wert
    - sonst $\Rightarrow$ undefiniert
- Zugriff auf Attribute
  - Allgemein:

```
(slot-value INSTANCE SLOTNAME)
```

  - bei Attributen mit *:accessor* angegebener Methodenname mit Instanz als Argument

```
(circle-center c)
```

Generische Funktionen (1)

Motivation:

- Klassen: *triangle, rectangle, circle*
- Gesucht: Funktion zur Flächenberechnung
- 1. Idee:

```
(defun area (figure)
   (cond ((typep figure 'triangle) ...)
         ((typep figure 'rectangle) ...)
         ((typep figure 'circle) ...)))
```

- 2. Idee: Definiere Funktionen *triangle-area, rectangle-area, circle-area*

Generische Funktionen (2)

- Generische Funktion = Menge von Methoden
- Methodenauswahl abhängig von Eingabeklasse
- Generische Funktion *area*:

```
(defmethod area ((figure triangle)) ...)
(defmethod area ((figure rectangle)) ...)
(defmethod area ((figure circle)) ...)
```

- Zusätzlich könnte man „else-Fall“ definieren:

```
(defmethod area (x) (error "Unzulässige Figur"))
```

- Es wird immer die spezifischste Methode gewählt.

Methodenkombination (1)

```
(defclass speaker () ())

(defmethod speak ((s speaker) string)
  (format t "~a" string))
```

Methode

Methodenkombination (1)

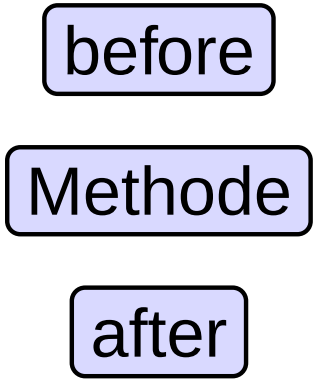
before

Methode

```
(defclass speaker () ())

(defmethod speak ((s speaker) string)
  (format t "~a" string))
(defmethod speak :before ((s speaker) string)
  (princ "I think "))
```

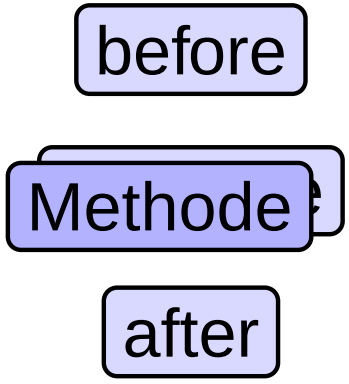
Methodenkombination (1)




```
(defclass speaker () ())

(defmethod speak ((s speaker) string)
  (format t "~a" string))
(defmethod speak :before ((s speaker) string)
  (princ "I think "))
```

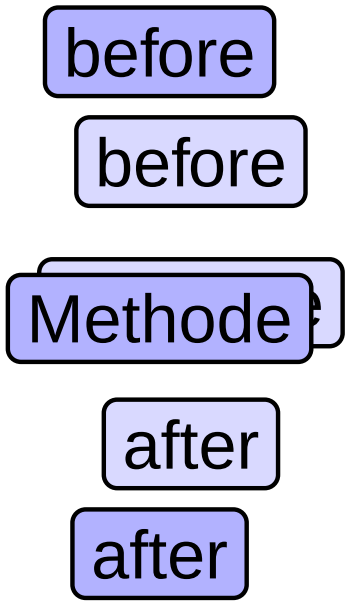
Methodenkombination (1)




```
(defclass speaker () ())
(defclass intellectual (speaker) ())

(defmethod speak ((s speaker) string)
   (format t "~a" string))
(defmethod speak :before ((s speaker) string)
   (princ "I think "))
```

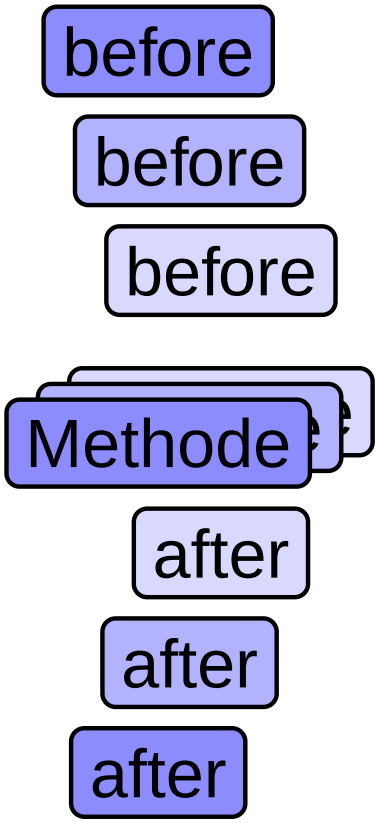
Methodenkombination (1)




```
(defclass speaker () ())
(defclass intellectual (speaker) ())

(defmethod speak ((s speaker) string)
   (format t "~a" string))
(defmethod speak :before ((s speaker) string)
   (princ "I think "))
(defmethod speak :before ((i intellectual) str)
   (princ "Perhaps "))
(defmethod speak :after ((i intellectual str))
   (princ " in some sense."))
```

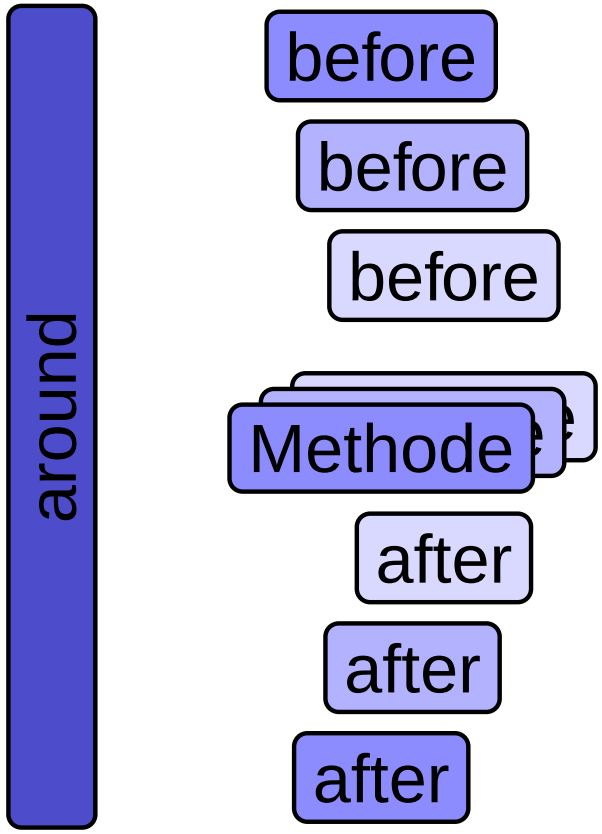
Methodenkombination (1)




```
(defclass speaker () ())
(defclass intellectual (speaker) ())

(defmethod speak ((s speaker) string)
   (format t "~a" string))
(defmethod speak :before ((s speaker) string)
   (princ "I think "))
(defmethod speak :before ((i intellectual) str)
   (princ "Perhaps "))
(defmethod speak :after ((i intellectual str))
   (princ " in some sense."))
```

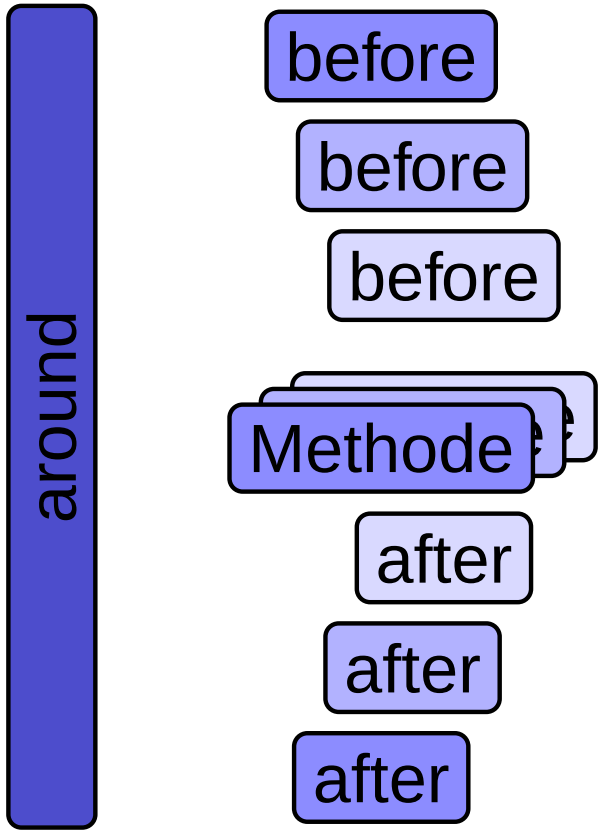
Methodenkombination (1)




```
(defclass speaker () ())
(defclass intellectual (speaker) ())

(defmethod speak ((s speaker) string)
   (format t "~a" string))
(defmethod speak :before ((s speaker) string)
   (princ "I think "))
(defmethod speak :before ((i intellectual) str)
   (princ "Perhaps "))
(defmethod speak :after ((i intellectual str))
   (princ " in some sense."))
```

Methodenkombination (1)




```
(defclass speaker () ())
(defclass intellectual (speaker) ())

(defmethod speak ((s speaker) string)
   (format t "~a" string))
(defmethod speak :before ((s speaker) string)
   (princ "I think "))
(defmethod speak :before ((i intellectual) str)
   (princ "Perhaps "))
(defmethod speak :after ((i intellectual str))
   (princ " in some sense."))
```

(speak (make-instance 'intellectual) "I'm hungry")

Methodenkombination (2)

- Kombination aller anwendbaren Methoden
- Operator wird mit *defgeneric* spezifiziert
- zweites Argument aller Methoden der generischen Funktionen muss der Operator sein
- Mögliche Operatoren:
  + *and append list max min nconc or progn*
- Kombination gilt nicht für "around-methods"

Methodenkombination (3)

Beispiel:

```
(defgeneric price (x)
   (:method-combination +))

(defclass jacket () ())
(defclass trousers () ())
(defclass suit (jacket trousers)) ())

(defmethod price + ((jk jacket)) 350)
(defmethod price + ((tr trousers)) 200)

(price (make-instance 'suit))
⇒ 550
```

Vergleich zu Java

	Java	Clos
Modell	Nachrichtenaustausch	generische Funktionen
Klasse	Attribute + Methoden	Attribute
Beispiel	*obj.move(10)*	*(move obj 10)*
	o1.conc(o2.conc(o3))	*(conc o1 (conc o2 o3))*

Das Nachrichtenaustauschmodell

- ist älter.
- ist Unterklasse des Modells der generischen Funktionen.
- kann durch generische Funktionen simuliert werden.