

6 Jenseits von Java

6.1 Variationen von Inheritance

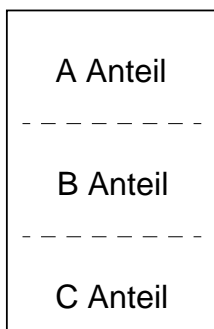
6.1.1 Konkatenation oder Delegation

Betrifft Implementierung von Objektspeicher und Methodenbindung

Konkatenation:

Objektspeicher:

zusammenhängender Speicher mit Anteilen für die Klasse und jede Oberklasse



Methodensuche:

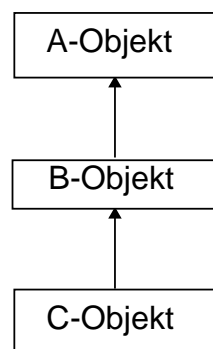
wird für C-Objekt vollständig erledigt

Sprachen: C++, Eiffel, Simula, Beta

Delegation:

Objektspeicher:

Speicherblock für das Objekt und Referenzen auf die Elternobjekte



Methodensuche:

Aufrufe werden in einem Objekt erledigt oder an Oberobjekt delegiert

Sprachen: Smalltalk, Self



6.1.2 Multiple Inheritance

Single Inheritance (Einfachvererbung):

Jede Klasse hat **höchstens oder genau eine direkte Oberklasse** (Base Class).

Inheritance-Relation ist linear aus Sicht einer Unterklasse;

Inheritance-Relation ist ein **Wald bzw. Baum** für alle Klassen zusammen.

Sprachen: Smalltalk, Java (für Klassen, nicht für Interfaces).

Multiple Inheritance (Mehrfachvererbung):

Jede Klasse hat **beliebig viele direkte Oberklassen**

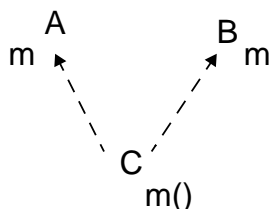
Inheritance-Relation ist DAG (aus Sicht einer Klasse und insgesamt)

Sprachen: C++, Eiffel, CLOS

Java's Inheritance und Interfaces können in C++ mit Multiple Inheritance und rein abstrakten Klassen nachgebildet werden.

C++ ermöglicht Übergang von Interfaces zu Klassen.

Namensproblem: gleicher Name in verschiedenen Klassen erreichbar



C++: undifferenzierter Zugriff **m** statt **A::m** ist Fehler

Eiffel: Erben ohne Umbenennung ist Fehler

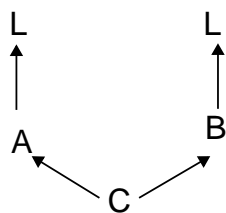
CLOS: Kanten sind geordnet, erster Weg zu **m** gilt

Mehrfach vorkommende Oberklasse

Eine Klasse **L** kann **mehrfach indirekte Oberklasse** einer Klasse **C** sein.

C++: zwei Varianten:

Konkatenation

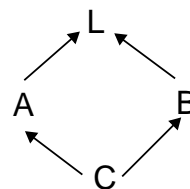


L-Anteil v. A.
A-Anteil
L-Anteil v. B
B-Anteil
C-Anteil

```
class A: public L {...}
class B: public L {...}
```

Objekte enthalten **L**-Anteile mehrfach
A und **B** benutzen **L** unabhängig
 (Attribute und Methoden)
C muss differenzieren

Eiffel: nur diese Variante mit expliziter Selektion



A-Anteil
B-Anteil
C-Anteil
L-Anteil

```
class A: public virtual L {...}
class B: public virtual L {...}
```

Objekte enthalten **L**-Anteil nur einmal;
L hat gemeinsame Daten für die Unterklassen

6.1.3 Selektives Erben in Eiffel

Einzelne Methoden und Datenattribute der Oberklasse können **beim Erben** explizit **überschrieben, umbenannt, gelöscht** werden:

- **Überschreiben (redefine)**
wird explizit gemacht!
- **Umbenennen (rename)**,
z. B. um Namenskonflikte zu vermeiden;
ermöglicht auch, dieselbe Klasse direkt mehrfach zu erben
Problem: Ersetzbarkeit, Subtyping werden zerstört!
- **Löschen** - nicht erben
undefine f end;
Problem: Ersetzbarkeit, Subtyping werden zerstört!
- Für **bestimmte** Klassen **zugreifbar** machen
export {A,B} f; ANY g; end

```
class FIXED_TREE[T] inherit
  TREE[T]
  redefine
    attach_to_higher, higher
  end;
  CELL[T];
  LIST [like Current]
  rename
    off as child_off, ...
  redefine
    duplicate, first_child
  end
  feature
    ....
  end;
```

Weitere Eigenschaften von Eiffel

- **Generische Definitionen**
- **Multiple inheritance:**
`FIXED_TREE[T]` erbt von
`TREE[T], CELL[T], LIST[...]`
- **Current** entspricht `this` o. `self`
- **like x:** steht für den Typ von `x`,
z.B. einsetzbar zur Lösung des Problems
der binären Operationen:

Eine Methode `plus` in der Klasse `A` sei
deklariert als

`like Current plus (like Current)..`

dann hat sie in `A` die Signatur

`plus: A x A -> A`

In einer Unterklasse `B` wird sie mit der
Signatur geerbt

`plus: B x B -> B`

```
class FIXED_TREE[T] inherit
    TREE[T]
    redefine
        attach_to_higher, higher
    end;

    CELL[T];

    LIST [like Current]
    rename
        off as child_off, ...
    redefine
        duplicate, first_child
    end

feature
    ....
end;
```

6.1.4 Mixin Inheritance

Mixin:

Wiederverwendbare Zusammenfassung weniger Methoden und Datenattribute für
bestimmten Zweck.

Mixins liegen **außerhalb der Klassenhierarchie**, werden von Klassen geerbt.

```
mixin Name
{ String name;
  void setName (String n){.... }
  string getName () {.... }
}
```

```
mixin Adresse
{.... }
```

```
class Person inherits Name, Adresse
{ .... }

class Firma inherits Adresse
{ .... }
```

Nicht explizit als Konstrukt in verbreiteten Sprachen.

Kann mit Mehrfachvererbung implementiert werden (z. B. in C++, Eiffel).

6.2 Varianten der dynamischen Methodenbindung Richtung der Methodensuche: aufwärts

Die Entscheidung, welche von **überschriebenen Methoden aufgerufen** wird, kann als **Suche entlang der Vererbungsrelation** beschrieben werden:

von der Klasse des Objekts **aufwärts durch die Oberklassen ersetzt** die überschreibende Methode die überschriebene **vollständig** - ggf. mit **anderer Bedeutung!**

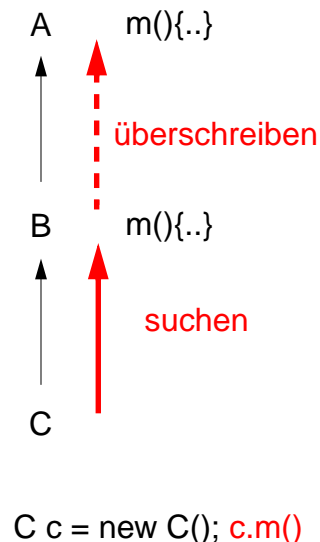
Die Oberklasse kann sich dagegen nicht absichern.

Sprachen: Java, C++ , Eiffel, Smalltalk

In Smalltalk ist die Suche so als Delegation implementiert.

In übersetzten Sprachen wird die Suche möglichst vermieden - durch Übersetzertabellen.

Wird auch **American semantics** genannt.



Richtung der Methodensuche: abwärts

Die Entscheidung, welche **Methoden aufgerufen** werden, kann als **Suche entlang der Vererbungsrelation** beschrieben werden:

Von der **höchsten Oberklasse abwärts zur Klasse des Objektes** werden **alle Methoden** gleichen Namens und passender Signatur **ausgeführt**.

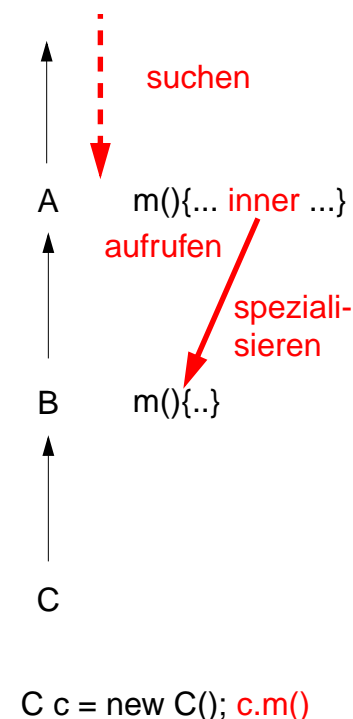
Im Rumpf der Methode kann durch **inner** gekennzeichnet werden, wo die Ausführung „tieferer“ Methoden eingebettet wird.

Die Oberklasse kann so **Erweiterungsstellen** definieren.

Die **Methoden** werden so in Unterklassen **spezialisiert** statt ersetzt.

Sprachen: Beta, Simula.

Wird auch **Scandinavian semantics** genannt.



Mehrfach-Dispatch

Einfach-Dispatch:

Methodenaufruf wird **auf ein Objekt** angewandt (Receiver): **a.plus (b)**

Die **Klassenzugehörigkeit des Objektes** in **a** bestimmt, welche Methode aufgerufen wird.
Die Parameterobjekte tragen nicht zu der Entscheidung bei.
Auch konzeptionell symmetrische Operationen werden **unsymmetrisch** ausgeführt.

Sprachen: fast alle objektorientierten Sprachen

Mehrfach-Dispatch:

Die **Klassenzugehörigkeiten aller Parameterobjekte zusammen** **plus (a, b)** bestimmen, welche Methode aufgerufen wird.

Symmetrische Behandlung der Parameter; kein hervorgehobener Receiver;
z. B. **plus** für **int**, **float** und **vector** unterschieden anhand der Objekte in **a** und **b**.

Sprachen: CLOS, Lisp-Derivate

6.3 Prototyp-basierte Sprachen

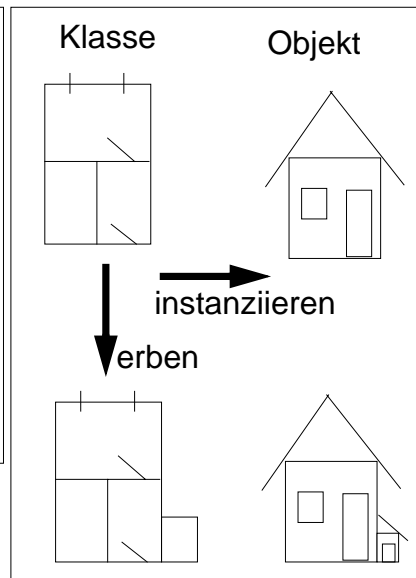
Klassen-basierte Sprachen (die meisten OO-Sprachen):

Klasse:

- Programmelement
- Plan zur Herstellung von Objekten
- vereinigt statische Eigenschaften der Objekte
- Vererbung zwischen Klassen

Objekt:

- zur Laufzeit erzeugt
- Exemplar einer Klasse
- speichert individuelle Daten, Zustand

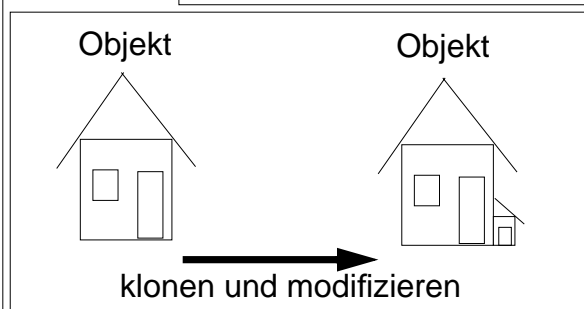


Prototyp-basierte Sprachen (Self, JavaScript):

Es gibt **keine Klassen** - nur **Objekte**.

Objekte werden

- **neu hergestellt** oder
- **aus Prototyp-Objekten kopiert** (geklont) und
- **individuell modifiziert**, weiterentwickelt
- **Vererbung zwischen Objekten**



JavaScript: Objektorientierte Scriptsprache

Scriptsprache:

- Sprache zum Aufruf von Programmen, ursprünglich Betriebssystem Kommandosprache, z. B. Unix Shell ähnlich: Perl, PHP, Python
- interpretiert
- dynamisch typisiert, einfache Datenstrukturen
- komfortable Operationen auf Strings

JavaScript:

- Scriptsprache zur Programmierung von Effekten auf Web-Seiten
- von Netscape entwickelt
- in Browser integriert
- kein Bezug zu Java
- objektorientiert, Prototyp-basiert
- in HTML (u.a.) integriert

```
<html>
  <head>
    <title>Beispiel mit JavaScript
    </title>
    <script type="text/javascript">
      function TestObjekt()
      { alert("Hello World!");
      }
    </script>
  </head>
  <form ...>
    <input type="submit" ...
      onclick="TestObjekt()">
  </form>
</body>
</html>
```

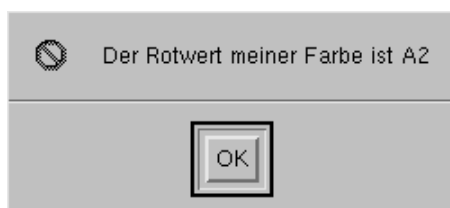
Objekterzeugung

Objekte werden durch **Konstruktorfunktionen** erzeugt:

- mit `new` aufgerufen,
- Funktionsrumpf greift über `this` auf Datenattribute zu
- **Datenattribute** werden **durch Zuweisung** eingeführt

```
function Farbe
  (Farbwert_R, Farbwert_G, Farbwert_B)
{ this.Farbwert_R = Farbwert_R;
  this.Farbwert_G = Farbwert_G;
  this.Farbwert_B = Farbwert_B;
}

function TestObjekt()
{ Test = new Farbe("A2", "FF", "74");
  alert("Der Rotwert meiner Farbe ist " +
    Test.Farbwert_R);
}
```

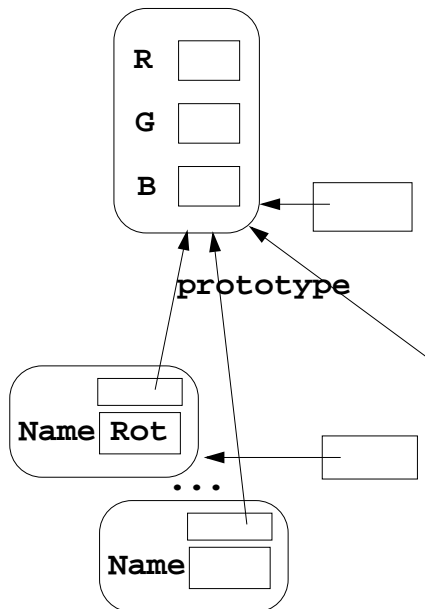


Objekt aus Prototyp erzeugen

Prototyp-Objekt der Konstrukturfunktion zuordnen.

Alle erzeugten Objekte erben die Prototyp-Eigenschaften.

Delegation: Ändern des Prototyps ändert alle Objekte dazu - auch existierende



```
function Farbe
  (Farbwert_R, Farbwert_G, Farbwert_B)
{  this.Farbwert_R = Farbwert_R;
   this.Farbwert_G = Farbwert_G;
   this.Farbwert_B = Farbwert_B;
}

Rot = new Farbe("FF", "00", "00");

function FarbeMitNamen(Name)
{  this.Name = Name;}

FarbeMitNamen.prototype = Rot;

function TestObjekt()
{  Test = new FarbeMitNamen("Rot");
   alert("Der Rotwert meiner Farbe ist " +
         Test.Farbwert_R +
         ". Der Name ist " + Test.Name);
}
```

Objekt-Hierarchie

Objekt-Hierarchie aus Prototypen; dynamische Zuordnung von Methoden:

Methode
zuweisen

Konstruktor-
Funktionen mit
fig als Prototyp

dynamische
Bindung für
Namen von
Attributen und
Methoden

veränderlicher
Zustand im
gemeinsamen
Prototyp

Überschreiben:
Suche entlang der
Prototyp-
Referenzen

```
fig = new function(){ }(); // leeres Objekt erzeugen
fig.move =
  function (dx, dy) // lambda-Ausdruck
  {  this.x += dx; this.y += dy};

function Kreis(r)
{  this.x = 0; this.y = 0; this.radius = r;}
Kreis.prototype = fig;

function Rechteck(l, b)
{  this.x = 0; this.y = 0; this.lg = l; this.br = b;}
Rechteck.prototype = fig;

function TestObjekt()
{  k1 = new Kreis(1); k1.move(5, 5);
   r1 = new Rechteck(3,7); r1.move(2,4);

   fig.printKoord =
     function ()
     {  alert( "x = " + this.x + " y = " + this.y);}

   k1.printKoord(); r1.printKoord();
}
```

Prototypen gegenüber Klassen

positiv für Prototypen:

- Prototypen sind **konkreter**, näher an Objekten als Klassen es sind.
- **Einzige Objekte** einer Art sind einfacher herzustellen.
- **Einfache**, freizügig verwendbare Sprachkonzepte
- Wenn **Klassen** „**first class**“ sind braucht man **Meta-Klassen** und dafür Meta-Klassen ...

negativ für Prototypen:

- **Konstruktorfunktionen** sind ein halber Schritt in Richtung Klassen.
- **Individuelle Objektzustände** können nicht im gemeinsamen Prototyp gespeichert werden; Abstraktion wird inkonsequent
- keine **Typsicherheit** (bezüglich Klassen)
- schwächere Möglichkeiten zur **Abstraktion**
- zur **Modellierung** nicht geeignet (Ebene dert Klassen und Typen fehlt)