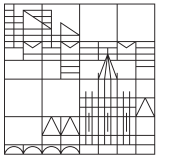
Universität Konstanz



Konzepte der Informatik

Programmierung Paradigmen

Barbara Pampel

Universität Konstanz, WiSe 2023/2024

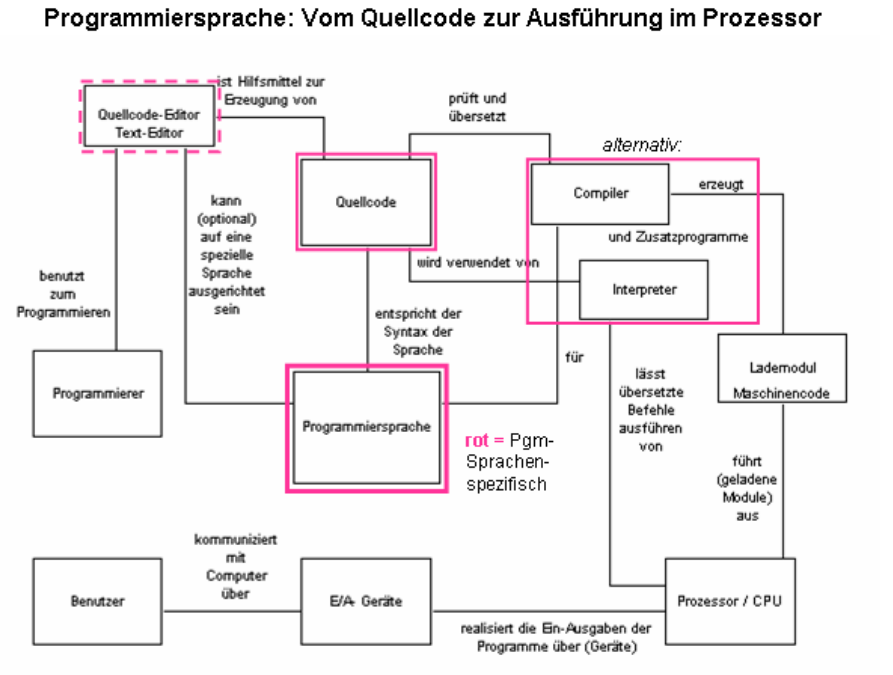
Inhalt

Was sind Programmiersprachen?

Wikipedia:

Eine Programmiersprache ist eine formale Sprache, die zur Formulierung von Computerprogrammen verwendet wird. Die Anweisungen (Befehle) können dabei entweder in Maschinencode geschrieben sein, der vom Prozessor ausgeführt wird, oder in Form eines abstrakteren, für Menschen besser lesbaren Quelltextes angegeben, der automatisiert in Maschinencode übersetzt werden kann.

Sprachübersetzung



Quelle: Wikipedia

Funktion und Ziel

- Maschinenlesbarkeit
  - effiziente Übersetzung in Maschinensprache
  - realisiert durch Einsatz von *kontextfreien* Sprachen
- Lesbarkeit durch den Menschen
  - Abstraktion von der Computerarchitektur
  - oft anlehnend an natürliche Sprache
- Abstraktion
  - von der internen Darstellung/Speicherung von Daten
  - von den Abläufen im Prozessor

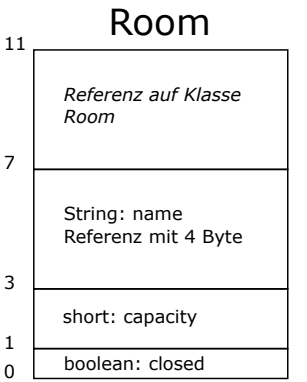
Fundamentale Datenabstraktion

- Computer kennt nur 1 Byte große Speicherplätze
- Aufteilung des Speichers in Programm-, Daten- und Stapelbereich
- Typisierung (und Zusammenfassung) von Speicherplätzen $\Rightarrow$ Datentypen
  - ganze Zahlen — `int` (4 Byte), `short` (2 Byte), `byte` (1 Byte)
  - Zeichen — je nach Zeichensatz 1, 2, 3 oder 4 Byte
  - Wahrheitswerte — 1 Byte (manchmal auch 2!)
- Speicherplatz bekommt erst durch Deklaration in der Programmiersprache Bedeutung zugewiesen
- Genaue Organisation im Speicher (z.B. *little endian* vs. *big endian*) für Programmierer uninteressant

Strukturierte Datenabstraktion

- Datenstruktur als fundamentales Konzept vieler Programmiersprachen
- Aufbau eines Objekts im Speicher für den Programmierer uninteressant

```
public class Room {
  private String name;
  private short capacity;
  private boolean closed;
}
```



Kontrollabstraktion

- Zuweisungen erfordern viele Schritte
  - `x = x + 3;`
  - Laden des Wertes der Variablen `x`
  - Addition mit Konstanten 3
  - Speichern des Ergebnisses in Variable `x`
- Bedingungen sind komplexe Abfolgen von
  - Werte laden und vergleichen
  - bedingte Sprünge durchführen
  - unbedingte Sprünge ans Ende des Bedingungsblockes
- Schleifen äquivalent
- Methoden sind nicht weiter gekennzeichnete Stellen im Maschinencode
  - Ausführen durch Anspringen der entsprechenden Adresse
  - Verwaltung des Stacks (Parameter, Rücksprungadresse, ...) für den Programmierer uninteressant

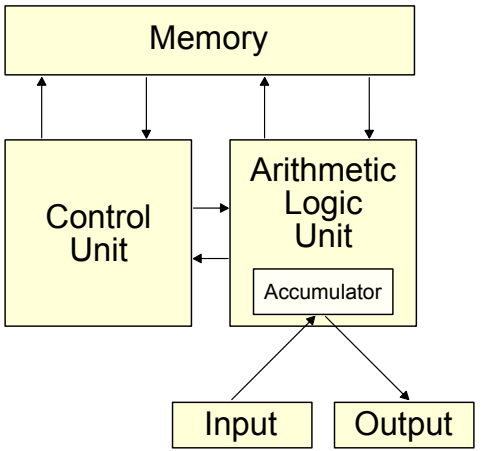
Warum viele Programmiersprachen?

- Kleine Skripte $\longleftrightarrow$ große Programmsysteme
- Einfache Verwendung $\longleftrightarrow$ komplexe Möglichkeiten
- Größe $\longleftrightarrow$ Geschwindigkeit
- Anwendungsgebiet
- Herkunft
  - „praxisnah” am konkreten Maschinenmodell, i.d.R. von-Neumann-Modell
  - traditionelle Mathematik basierend auf Funktionen
  - symbolische Logik aus dem 19. Jahrhundert

Inhalt

von-Neumann-Modell

- zentraler Speicher
- Kontrolleinheit mit sequentieller Ausführung von Befehlen



Charakterisierung

- **imperare** lateinisch für *befehlen, anweisen*
  $\Rightarrow$ **Anweisung** als zentrales Konstrukt
- **Kontrollstrukturen** zur Steuerung der Befehlsausführung
- **Datenstrukturen** zur Organisation von Daten
- **Funktionen und Prozeduren** zur Strukturierung
  $\Rightarrow$ (prozedurale Programmierung)

Atomare Anweisungen und Ausdrücke

- Zuweisungen
- Operatoren
- Vergleiche
- Prozedur- und Funktionsaufrufe

Atomare Anweisungen und Ausdrücke

- Zuweisungen

Beispiel in Pascal

```
pos := 0;
```

- Operatoren
- Vergleiche
- Prozedur- und Funktionsaufrufe

Atomare Anweisungen und Ausdrücke

- Zuweisungen
- Operatoren

Beispiel in Pascal

```
pos := pos + 1;
```

- Vergleiche
- Prozedur- und Funktionsaufrufe

Atomare Anweisungen und Ausdrücke

- Zuweisungen
- Operatoren
- Vergleiche

Beispiel in Pascal

```
pos = n
```

$\Rightarrow$ Ausdruck vom Typ

```
boolean
```

- Prozedur- und Funktionsaufrufe

Atomare Anweisungen und Ausdrücke

- Zuweisungen
- Operatoren
- Vergleiche
- Prozedur- und Funktionsaufrufe

Beispiel in Pascal

```
Write('found at position ', pos);
```

oder eigene

```
LinSearch(d,g);
```

Kontrollstrukturen

- Sequenz
- Bedingungen/Verzweigungen
- Schleifen
- Sprünge

Kontrollstrukturen

- Sequenz

Beispiel in Pascal

```
begin
stat := 'searching';
pos := 0;
pos := pos +1;
end.
```

- Bedingungen/Verzweigungen
- Schleifen
- Sprünge

Kontrollstrukturen

- Sequenz
- Bedingungen/Verzweigungen

Beispiel in Pascal

```
if gesucht = d[pos] then stat := found
else
  if pos = n then stat := 'end of data';
```

- Schleifen
- Sprünge

Kontrollstrukturen

- Sequenz
- Bedingungen/Verzweigungen
- Schleifen

Beispiel in Pascal

```
for i: 1 to n do
   if gesucht = dataArray[i] then stat := 'found';
```

- Sprünge

Kontrollstrukturen

- Sequenz
- Bedingungen/Verzweigungen
- Schleifen

Beispiel in Pascal

```
repeat
   pos := pos + 1;
   if gesucht = dataArray[pos] then stat := 'found'
   else
      if pos = n then stat := 'end of data'
until CompareText('searching',stat) <> 0;
```

- Sprünge

Kontrollstrukturen

- Sequenz
- Bedingungen/Verzweigungen
- Schleifen
- Sprünge

Beispiel in Pascal

```
Label searchOn;
searchOn:
pos := pos + 1;
if gesucht <> dataArray[pos] then
goto searchOn;
```

Aber:

Edsger W. Dijkstra: Letters to the editor: Go To Statement Considered Harmful. In: Communications of the ACM. 11, Nr. 3, März 1968, S. 147 - 148

Programm-Beispiel in Pascal

```
Program test;
Uses sysutils, crt;
Const n= 10;
type data = array [1..n] of integer;

Procedure LinSearch (var dataArray : data; gesucht : integer);
var stat : string; pos : integer;
begin
stat := 'searching';
pos := 0;
repeat
pos := pos + 1;
if gesucht = dataArray[pos] then stat := 'found'
else
    if pos = n then stat := 'endofdata';
```

Programm-Beispiel in Pascal

```
...
writeln(stat);
until CompareText('searching',stat) <> 0;
end;

var g,i: integer;
d: data;
begin
ClrScr;
for i:= 1 to n do d[i] := i;
g := 8;
LinSearch (d, g);
end.
```

Vor- und Nachteile

- Vorteile
  - leicht zu verstehen durch schrittweise Ausführung von Anweisungen
- Nachteile
  - Seiteneffekte — Methoden können Zustand des Programms (ungewollt) ändern
  - Sequentielle Ausführung — verteilte Ausführung auf mehreren Prozessoren nicht einfach zu realisieren (von-Neumannscher Flaschenhals)

Fortran 77

- 1957 John Backus von IBM: Sprache zur ***For**mula **tran**slation*
- bis heute im Einsatz bei numerischen Berechnungen (Physik, Chemie, ...)
- Nur ein einfache Zählschleife

```
PROGRAM ALT
      IMPLICIT NONE
      REAL X, SUMME
C
      WRITE(*,*)
     *    'Das Programm addiert die von Ihnen eingegebenen Zahlen'
      WRITE(*,*)
      WRITE(*,*) 'Wollen Sie die Summation abschliessen,'
      WRITE(*,*) 'so geben Sie als Zahl 0 ein'
      WRITE(*,*)
C
      SUMME = 0.0
      X = 0.0
C
100   CONTINUE
      SUMME = SUMME + X
      WRITE(*,'(1X,A$)') 'Geben Sie die Zahl ein: '
      READ(*,*) X
      IF ( X .NE. 0.0) GOTO 100
C
      WRITE(*,*)
      WRITE(*,*) 'Die Summe betraegt: ', SUMME
      END PROGRAM
```

Fortran 90/95

- Fortran 90/95 ist wesentlich komfortabler
  - „freie" Eingabe von Quellcode (Groß-/Kleinschreibung, Leerzeichen, ...)
  - rekursive Funktionen
  - Bezeichner mit bis zu 31 Zeichen
  - Schleifen mit Abbruchbedingung

```
program neu
   implicit none
   real :: x = 0.0, summe = 0.0

    write(*,*) 'Das Programm addiert die von Ihnen eingegebenen Zahlen'
    write(*,*)
    write(*,*) 'Wollen Sie die Summation abschliessen, so geben Sie als Zahl 0 ein'
    write(*,*)

schleife: do
    summe = summe + x
    write(*,'(1X,A$)') 'Geben Sie die Zahl ein: '
    read(*,*) x
    if ( x /= 0.0) cycle schleife
    exit schleife
end do schleife

    write(*,*)
    write(*,*) 'Die Summe betraegt: ', summe

end program neu
```

ALGOL I

- 1958 **Algo**rithmic **L**anguage
- Basis für fast alle späteren imperativen Programmiersprachen
- Blockstruktur mit explizitem Beginn und Ende
- Unterschiedliche Definitionen für Referenz, Publikation und Implementierung

```
procedure Absmax(a) Size:(n, m) Result:(y) Subscripts:(i, k);
    value n, m; array a; integer n, m, i, k; real y;
comment The absolute greatest element of the matrix a, of size
n by m is transferred to y, and the subscripts of this element
to i and k;
begin integer p, q;
  y := 0; i := k := 1;
  for p:=1 step 1 until n do
  for q:=1 step 1 until m do
    if abs(a[p, q]) > y then
       begin y := abs(a[p, q]);
       i := p; k := q
       end
end Absmax
```

Pascal

- Benannt nach dem Mathematiker Blaise Pascal
- Entwickelt 1970 von Niklaus Wirth, primär zum Einsatz in der Lehre
- Weiterentwicklung von ALGOL 60
- Benutzerdefinierte Datenstrukturen aufbauend auf einfachen Variablen und Reihungen
- Starke Typisierung von Datentypen, d.h. keine automatische Konvertierung
- Ziele: schnelle Übersetzung und schnelle Programme
- Konzept des *P-Codes*, Vorläufer des Java Byte Codes

C

- Entwickelt 1972 von Dennis Ritchie zum Einsatz in Unix-Betriebssystemen
- Eine der populärsten Programmiersprachen
- Ziele
  - einfacher Übersetzer
  - maschinennahe (effiziente) Programmierung
  - minimales Laufzeitsystem
  - Portabilität
- Syntax praktisch identisch zu Java
- Verbreitete Verwendung von Zeigern
- 1978 erweiterte Version von Brian Kernighan und Dennis Ritchie
- Heute standardisiert als ANSI/ISO C

C

```c
#include <stdio.h>
int main() {
int x,y,z,sum;
x = 1;
y = 2;
z = 4;
sum = x + y + z;
/* Summe ausgeben */
printf("summe = %d \n", sum);
return 0;
}
```

Inhalt

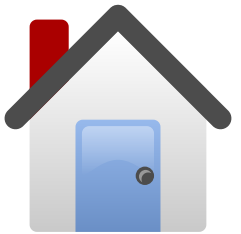
Objektorientierte Programmierung

- Warum überhaupt objektorientierte Programmierung?

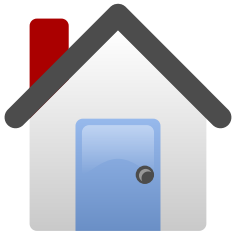
Objektorientierte Programmierung

- Warum überhaupt objektorientierte Programmierung?
- Problem (siebziger Jahre lt. Jhdt.)
  - Programme werden sehr groß (UNIX, Datenbanken, Informationssysteme etc.)
  - immer mehr Varianten desselben Programms (verschiedene Versionen, Portierungen auf immer mehr Geräte nötig, Vielfalt an Umgebungen und Randbedingungen entsteht)
  - wie verwaltet man das?
- Simula (ab 1964)
  - Sprache für physikalische Simulation
  - Simulationen im Schiffsbau, viele Varianten und Parameter
  - auch hier: unüberschaubare Programmvarianten
  - Idee: führe Objekte ein, die Zustände haben und über Nachrichten kommunizieren
  - später: Vererbung, Ableitung, Klassen

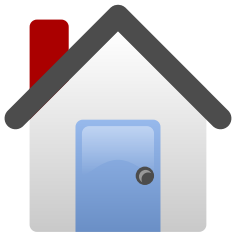
Was ist ein Objekt?



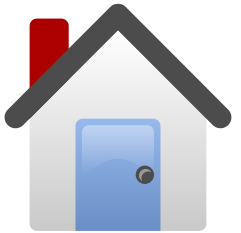
Was ist ein Objekt?



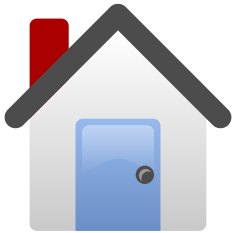
Was ist ein Objekt?



Was ist ein Objekt?



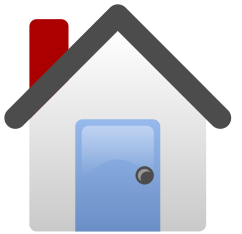
Was ist ein Objekt?



- Alles ist ein Objekt!

Objekteigenschaften I

- Objekte haben Eigenschaften oder einen Zustand



- Farbe = grau
- Anzahl Türen = 1
- Anzahl Fenster = 0
- Flachdach? = nein

Objekteigenschaften II

- Objekte haben Eigenschaften oder einen Zustand

- Farbe = blaugrün
- Höchstgeschwindigkeit = 150 $\frac{km}{h}$
- Geschwindigkeit = 30 $\frac{km}{h}$
- Fahrgestellnummer = AX76TGHA

Objekteigenschaften II

- Objekte haben Eigenschaften oder einen Zustand

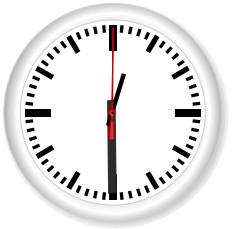
  - Farbe = blaugrün
  - Höchstgeschwindigkeit = 150 $\frac{km}{h}$
  - Geschwindigkeit = 30 $\frac{km}{h}$
  - Fahrgestellnummer = AX76TGHA

- Veränderliche Eigenschaften
  - Farbe, aktuelle Geschwindigkeit, Uhrzeit, Akkuladestand, ...
- Unveränderliche Eigenschaften (Naja, fast)
  - Flachdach?, Fahrgestellnummer, Anzahl Zeiger, Bildschirmdiagonale

Nachrichten an Objekte I

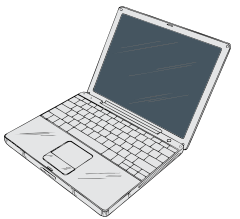
- *Setze Uhrzeit auf 12:30!*

⇒

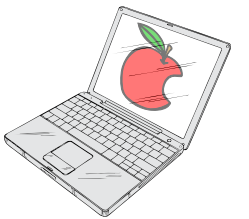


Nachrichten an Objekte II

- *Anschalten!*



⇒



Nachrichten an Objekte III

- *Um 20 $\frac{km}{h}$ beschleunigen!*

$\Rightarrow$

Nachrichten an Objekte III

- *Um 20 $\frac{km}{h}$ beschleunigen!*

$\Rightarrow$

- Unterschiedliche Aktionen auf Nachrichten
  - Einfaches Setzen von Eigenschaften (Uhrzeit, Farbe, ...)
  - Überprüfen der Eingabe und dann verändern von Eigenschaften (Geschwindigkeit $\Leftrightarrow$ Höchstgeschwindigkeit)
- Nachrichten durch Aufruf von Methoden

Nachrichten an Objekte IV

- Methoden/Nachrichten können parametrisiert sein
  - neue Farbe
  - neue Uhrzeit
  - neue Geschwindigkeit
- ... oder unparametrisiert
  - Anschalten!
  - Vollbremsung!
  - Licht an!
- Methoden können Ergebnisse zurück liefern, sog. Rückgabewerte
  - neue Geschwindigkeit nach dem Beschleunigen
  - Hat das Anschalten funktioniert?
  - Batterieladestand

Objektkomposition I

- Objekte können aus anderen Objekten zusammengesetzt sein

Sekundenzeiger

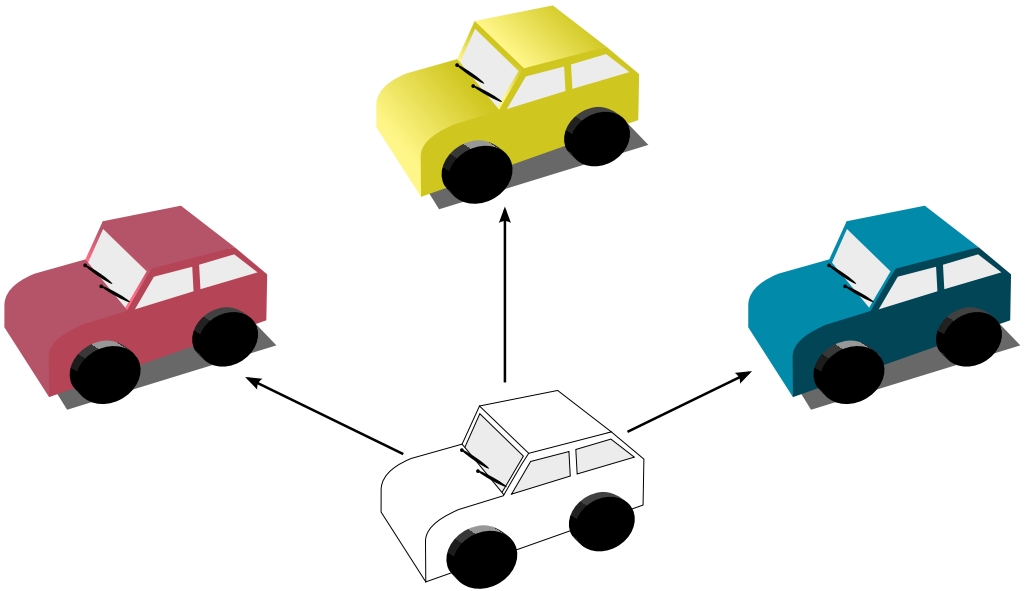
Minutenzeiger

Stundenzeiger

Ziffernblatt

Klassen I

- Klassen sind Typen von Objekten
- Klasse *Auto* mit verschiedenen Objekten und verschiedenen Eigenschaften



Klassen II

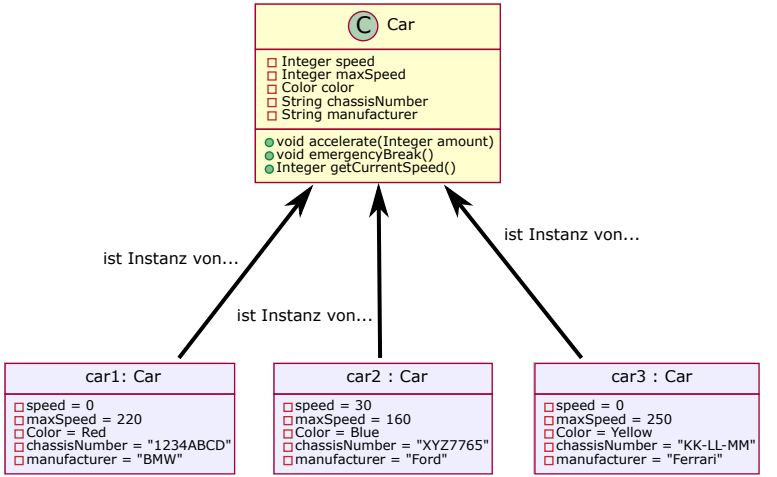
- Klasse *Uhr* mit verschiedenen Objekten und verschiedenen Eigenschaften

Klassen und Objekte I

- Von einer Klasse können Instanzen erzeugt werden
- Klassen definieren Eigenschaften oder Instanzvariablen, aber *keine* Werte
  - Farbe - Farbe setzen
  - Geschwindigkeit - beschleunigen
  - Batterieladestand - anschalten
- Objekte füllen die Instanzvariablen mit konkreten Werten $\Rightarrow$ Zustand
  - Uhrzeit = 10:10 — Uhrzeit = 10:29
  - Farbe = blaugrün — Farbe = rot — Farbe = gelb
  - Stockwerke = 1 — Stockwerke = 20
- Klassen definieren Methoden, die den Zustand von Objekten ändern (können)
- Klassen als Schablonen

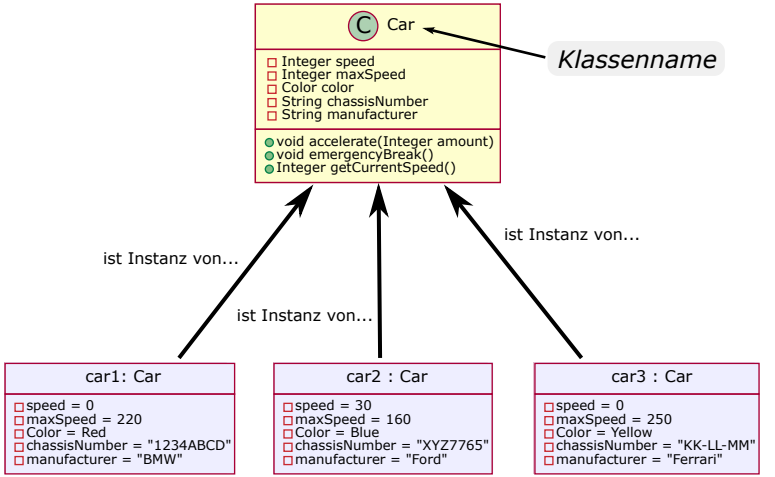
Klassen und Objekte II

- Abstraktere Darstellung, sog. Klassendiagramm
  - Teil der UML (Unified Modeling Language)



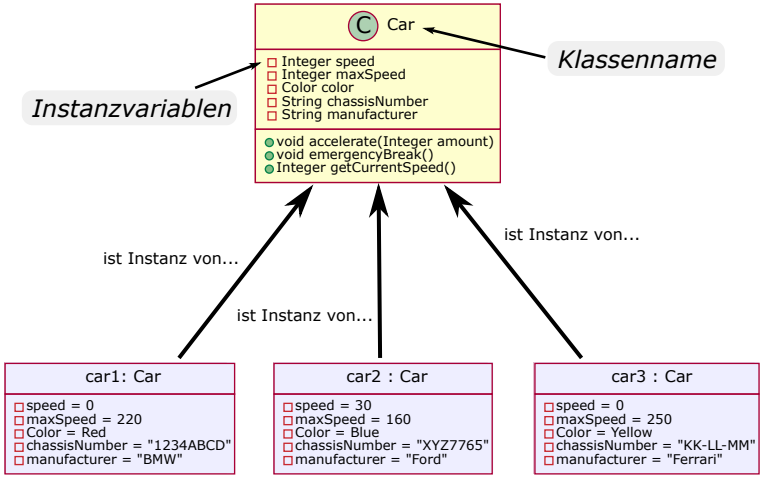
Klassen und Objekte II

- Abstraktere Darstellung, sog. Klassendiagramm
  - Teil der UML (Unified Modeling Language)



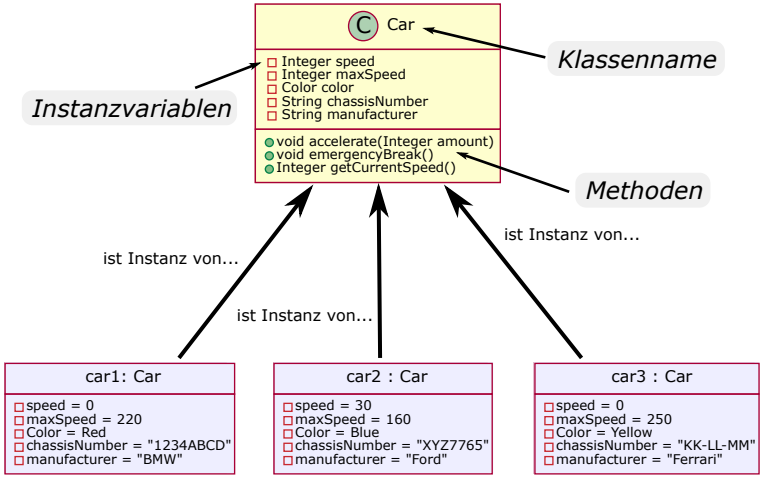
Klassen und Objekte II

- Abstraktere Darstellung, sog. Klassendiagramm
  - Teil der UML (Unified Modeling Language)



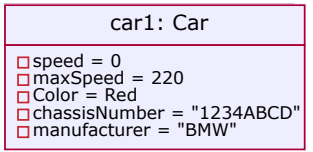
Klassen und Objekte II

- Abstraktere Darstellung, sog. Klassendiagramm
  - Teil der UML (Unified Modeling Language)



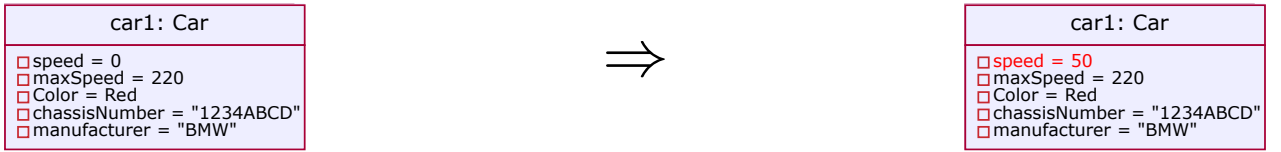
Methodenaufrufe I

```
car1.accelerate(50);
```



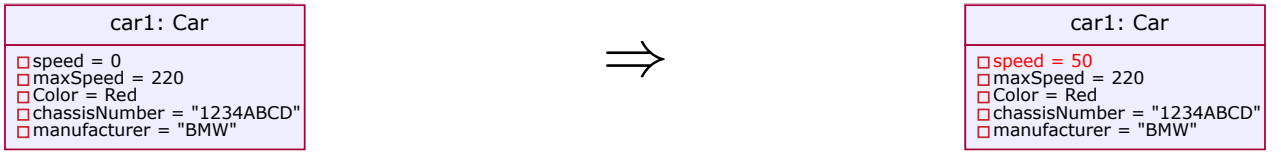
Methodenaufrufe I

```
car1.accelerate(50);
```

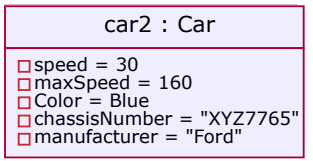


Methodenaufrufe I

```
car1.accelerate(50);
```

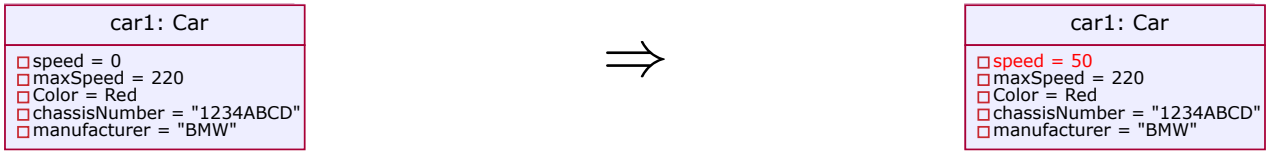



```
car2.emergencyBreak();
```

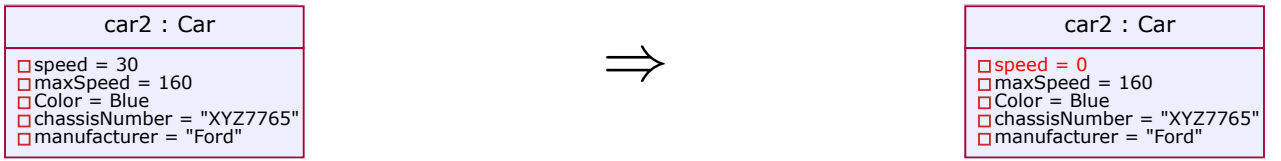


Methodenaufrufe I

```
car1.accelerate(50);
```

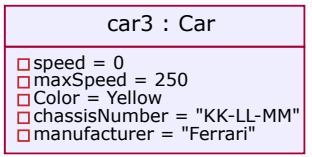



```
car2.emergencyBreak();
```



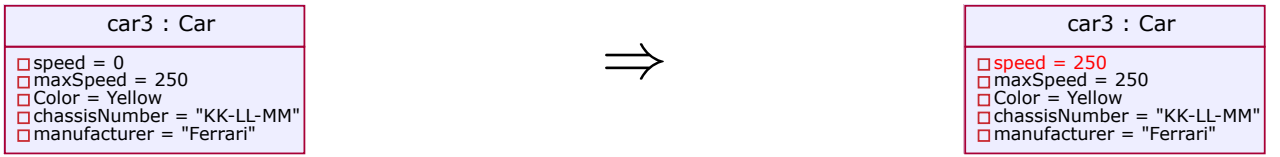
Methodenaufrufe II

```
car3.accelerate(300);
```



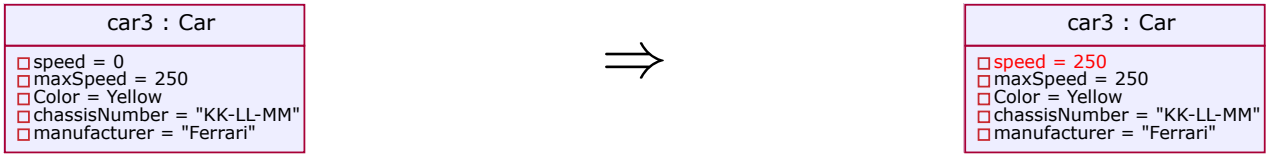
Methodenaufrufe II

```
car3.accelerate(300);
```



Methodenaufrufe II

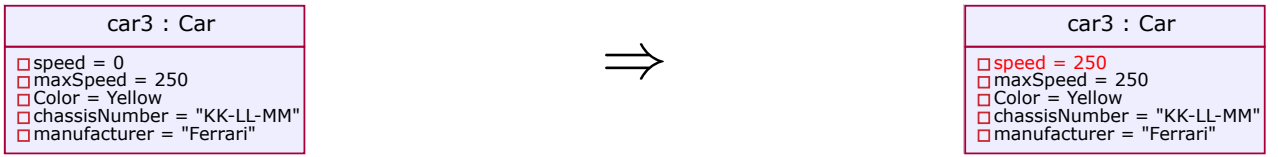
```
car3.accelerate(300);
```




```
car3.getCurrentSpeed();
```

Methodenaufrufe II

```
car3.accelerate(300);
```




```
car3.getCurrentSpeed();
```
- 250

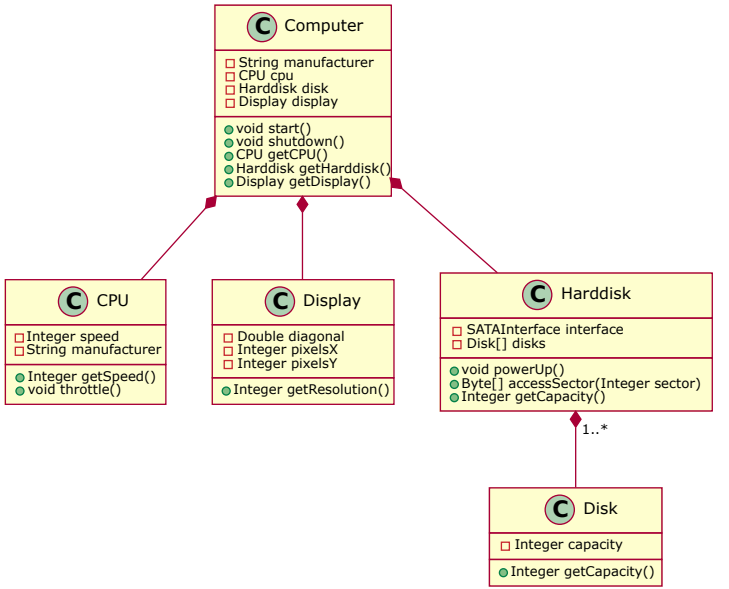
Beobachtungen

- Zustand eines Objekt lässt sich (nur) durch Methodenaufrufe ändern
- Methodenaufrufe ändern nur Zustand des Objekts, auf dem sie aufgerufen werden
- Methoden machen oft mehr, als nur Instanzvariablen zu setzen
- Bestimmte Eigenschaften können nicht verändert werden
- Eigenschaften können (nur) durch Methodenaufrufe abgefragt werden
- Was fehlt in unserem Beispiel?

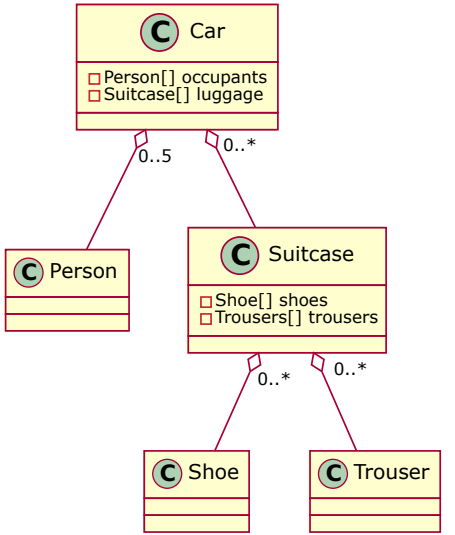
Assoziationen zwischen Klassen

- Objekte können aus anderen Objekten zusammengesetzt sein
  - Teil-Ganzes-Beziehung
  - Ganzes kann nicht ohne das Teil existieren
  - Komposition auf Klassenebene
- Objekte können lose mit anderen Objekten verbunden sein
  - Ganzes kann ohne das Teil existieren
  - Aggregation auf Klassenebene

Komposition



Aggregation



Beispiele – C++

- Erste Version 1983 von Bjarne Stroustrup
- Erweiterung von C durch Klassen, Überladung, Templates und Ausnahmen
- Seit 1998 ISO-Standard
- Unterstützung durch C++ Standard Library
  - Kombination der Standard Template Library und der C Library
- Eine der meistverwendeten Programmiersprachen
- Probleme: sehr komplex, keine automatische Speicherverwaltung

Beispiele – C++

```
class Studycourse {
  private:
    std::string title;
    Lecturer& lecturer;
    Room& room;
    std::vector<Student&> students;
  public:
    void add(Student& object) { ... }
    virtual int getSuccessfulStudents() = 0;
}
```

Inhalt

Funktionale Programmierung

- (Mathematische) Funktion als zentrales Sprachmittel
- In Reinform keine Zuweisungen und keine Variablen!

Funktionale Programmierung

- (Mathematische) Funktion als zentrales Sprachmittel
- In Reinform keine Zuweisungen und keine Variablen!
- keine Schleifen, deswegen **Rekursion** als zentrales Konzept:

math. Definition der Fakultäts-Funktion

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{sonst} \end{cases}$$

in Haskell

```
fac 0 = 1
fac n = n * fac (n-1)
```

Funktionale Programmierung

- (Mathematische) Funktion als zentrales Sprachmittel
- In Reinform keine Zuweisungen und keine Variablen!
- keine Schleifen, deswegen **Rekursion** als zentrales Konzept:

math. Definition der Fakultäts-Funktion

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{sonst} \end{cases}$$

in Haskell

```
fac 0 = 1
fac n = n * fac (n-1)
```

- Ergebnis einer Funktion hängt ausschließlich von den Parametern ab (kein innerer Zustand)
- Funktionen können – wie Werte – als Parameter und Rückgabewerte verwendet werden

Funktionale Programmierung

- Vorteile
  - keine Seiteneffekte
  - leicht zu parallelisieren
  - automatische Speicherverwaltung einfach
- Nachteile
  - ungewohnte Schreibweise
  - teilweise ineffizient

Beispiel – Scheme I

- Basierend auf LISP
  - entwickelt 1958 von John McCarthy am MIT
  - Basisstruktur Liste, verwendet für Programm *und* Daten
  - Interpreter ist selbst in Lisp geschrieben
- Abgespeckte Version von Lisp, entwickelt von Guy Steele und Gerald Sussman am MIT
- Dynamische Typisierung
  - Datentypen werden erst zur Laufzeit überprüft
- Erstmals verwendet 1975, hauptsächlich in Forschung und Lehre
- IEEE-Standard

Beispiel – Scheme II

- Definition einer Funktion mit einem Parameter
- Funktionsaufrufe in Präfixnotation (Funktionsname zuerst)
- `lambda` erzeugt eine anonyme Funktion
- `define` weißt Lambda-Ausdrücken einen Namen zu

```
(define add1 (lambda (x) (+ x 1)))
(define sub1 (lambda (x) (- x 1)))
(define square (lambda (x) (* x x)))
```

Beispiel – Scheme III

- Funktion als Parameter
  - wiederholte Anwendung einer Funktion

```
(define (repeat f n)
    (cond ((= n 0) (lambda (x) x))
          ((> n 0) (lambda (x)
                     ((repeat f (- n 1)) (f x))
                   )
    )   )     )
```

Beispiel – Scheme III

- Funktion als Parameter
  - wiederholte Anwendung einer Funktion

```
(define (repeat f n)
    (cond ((= n 0) (lambda (x) x))
          ((> n 0) (lambda (x)
                     ((repeat f (- n 1)) (f x))
                   )
    )   )     )
```

- Viermaliges Quadrieren (x^{16})

```
(define (pot16 x) ((repeat square 4) x))
(square (square (square (square x))))
```

Beispiel – Miranda I

- Erste Verwendung 1985, entwickelt von David Turner
- Funktionale Sprache mit Algol-ähnlicher Syntax
- Funktionsdefinition oft mit *Musterkennung* (*pattern matching*)

```
fact 0 = 1
fact n = n * fact (n - 1)

sum [] = 0
sum (a:x) = a + sum x
```

- Statische Typisierung mit Typinferenz
  - Datentypen können automatisch ermittelt werden

Beispiel – Miranda II

- *Lazy Evaluation*
  - Auswertung von Funktionen nur, wenn sie wirklich benötigt werden
  - in den meisten imperativen Sprachen nicht oder nur in Spezialfällen möglich

```
cond True x y = x
cond False x y = y

cond (x = 0) 0 (1 / x)
```

- In Java in etwa

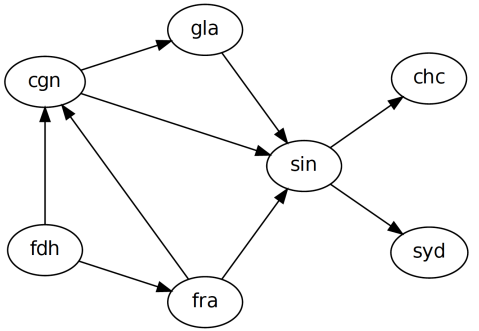
```
Object cond(boolean b, Object ifTrue, Object ifFalse) {
  if (b) { return ifTrue; } else { return ifFalse; }
}
cond(x == 0, 0, 1 / x);
```

Inhalt

Logische Programmierung I

- Basierend auf automatischen Theorembeweisern
- Verwendung von Konzepten aus der formalen Logik
  - Konstanten
  - Prädikate, z.B. *even(n)* oder *greater-zero(x)*
  - Funktionen, z.B. *sucessor(n)*
  - Variablen
  - Junktoren (ähnlich den bekannten booleschen Operatoren)
  - Quantoren, z.B. Existenzquantor $\exists$ und Allquantor $\forall$
- Bekanntester Vertreter Prolog
- Entwickelt 1972 von Alain Colmerauer
- Vorteile
  - Lösung von Problemen ohne explizite Beschreibung
- Nachteile
  - langsam

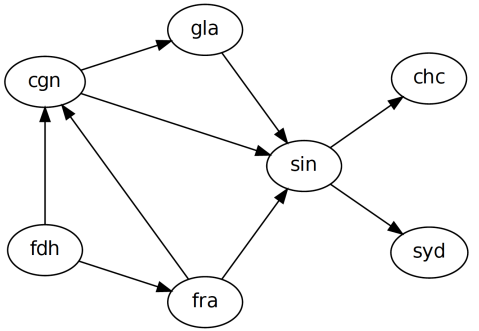
Beispiel Prolog (Aus der Vorlesung *Konzepte der Programmierung*)



Prolog Fakten

```
leg(fdh, cgn). leg(fra, cgn).
leg(cgn, sin). leg(fdh, fra).
leg(sin, syd). leg(fra, sin).
leg(cgn, gla). leg(gla, sin).
leg(sin, chc).
```

Beispiel Prolog (Aus der Vorlesung *Konzepte der Programmierung*)



Prolog Fakten

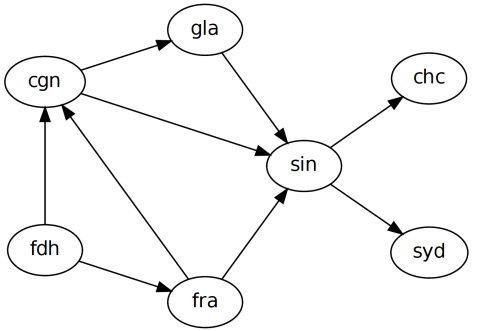
```
leg(fdh, cgn). leg(fra, cgn).
leg(cgn, sin). leg(fdh, fra).
leg(sin, syd). leg(fra, sin).
leg(cgn, gla). leg(gla, sin).
leg(sin, chc).
```

Prolog Regeln

```
route(A, B, []) :-
leg(A, B).

route(A, B, [X|XS]) :-
leg(A, X),
route(X, B, XS).
```

Beispiel Prolog (Aus der Vorlesung *Konzepte der Programmierung*)



Prolog Fakten

```
leg(fdh, cgn). leg(fra, cgn).
leg(cgn, sin). leg(fdh, fra).
leg(sin, syd). leg(fra, sin).
leg(cgn, gla). leg(gla, sin).
leg(sin, chc).
```

Prolog Regeln

```
route(A, B, []) :-
leg(A, B).

route(A, B, [X|XS]) :-
leg(A, X),
route(X, B, XS).
```

Prolog Aufruf

```
?- route(fra, sin, Q).
Q = [] ;
Q = [cgn] ;
Q = [cgn, gla] ;
false.
```

Inhalt

Parallele Programmierung

- Aufgaben-/ Programmteile werden in separaten Prozessen/Threads ausgeführt

⇒ Nebenläufigkeit

- können dann unabhängig von mehreren Prozessoren gleichzeitig ausgeführt werden

⇒ Parallelisierung

Parallelisierungsstrategien

Datenparallelisierung
- Zu verarbeitenden Daten können so aufgeteilt werden, dass sie unabhängig von mehreren Prozessoren verarbeitet werden können
- Prozessoren müssen nicht zwangsläufig exakt die gleichen Operationen ausführen
- englisch *data parallel algorithms*

Aufgabenparallelisierung
- Verschiedene Prozesse/Threads bearbeiten unterschiedliche Schritte des Problems
- Daten können identisch sein, müssen aber nicht
- Prozesse tauschen oft (Zwischen)ergebnisse aus

Parallelisierungsstrategien

- Parallelsprachen
  - optimiert für nebenläufige Ausführung von Programmteilen
  - Mechanismen, um „gleichzeitigen” Zugriff auf Speicherbereiche zu koordinieren
  - Beispiele: Ada, Clojure, E, occam, Cilk, ...
- die meisten Programmiersprachen ermöglichen Parallelisierung von Abläufen
  - C, C++, Java (explizite Parallelisierung)
  - pH. parallel Haskell (implizite Parallelisierung)

Inhalt

Weitere Kategorien I

- Vektorsprachen, auch multidimensionale Sprachen
  - Methoden/Operationen werden auf Vektoren (oder Reihungen) von Objekten simultan angewendet

```
> x <- c(1, 2, 3, 4, 5, 6)
> y <- x^2
> print(y)
[1] 1 4 9 16 25 36
```

  - Beispiele: F, Fortran 90, IDL, MATLAB, R, ZPL, ...

Weitere Kategorien II

- Assemblersprachen
  - Befehle entsprechen direkt den vom Prozessor unterstützen Instruktionen
  - architekturspezifisch

```
; Summe von 1 bis Wert an der Adresse 0x12345678
        mov ecx, (0x12345678)
        mov eax, 0
label:  add eax, ecx
        dec ecx
        jnz @label
        mov (0x1234567C), eax
```

Weitere Kategorien III

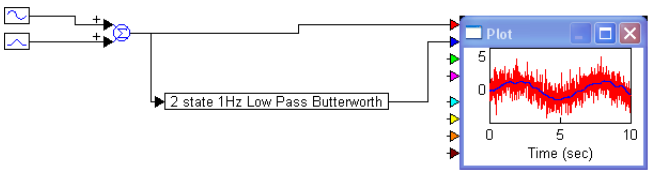
- Kommandozeilensprachen, auch Batch-Sprachen
  - zur Automatisierung von Aufgaben im Betriebssystem
  - interaktive Ausführung der Befehle möglich

```
for i in *.txt; do
  if [ $i -ot new.txt ]; then
    echo $i;
  fi
done
```

  - Beispiele: bash, csh, tcsh, cmd, zsh, ...

Weitere Kategorien IV

- Datenflusssprachen
  - meist visuelle Darstellung von Bearbeitungsschritten für eingehende Daten



  - Beispiele: G, Lucid, VisSim, VHDL, ...
- Datenorientierte Sprachen

Weitere Kategorien V

- i.d.R. Suche und Bearbeitung von in Relationen oder Tabellen gespeicherten Daten

```
UPDATE students SET credits = credits + 10 INNER JOIN
  exams ON (exams.student = students.id)
  WHERE (exams.grade <= 4.0) AND (exams.name = 'KdI')
```

  - Beispiele: SQL, Clipper, dBase, Visual FoxPro, WebQL
- Deklarative Sprachen
  - Beschreibung eines Problems anstatt der Lösung
  - Sprache „findet” die Lösung
  - Beispiele: SQL, Prolog, Analytica, ...

Weitere Kategorien VI

- Esoterische Sprachen
  - ohne praktischen Nutzen, oft zur Demonstration eines Konzepts oder zur Erheiterung

```
 // Addition zweier Ziffern
,>++++++[<-------->-],[<+>-]<.

// Multiplikation zweier Ziffern
,>,>++++++++[<------<------>>-]
<<[>[>+>+<<-]>>[<<+>>-]<<<-]
>>>++++++[<+++++++++>-],<.>.
```

  - Beispiele: Befunge, Brainfuck, Shakespeare, Whitespace, ...

Weitere Kategorien VII

- Reflektive Sprachen
  - Möglichkeit, die Programmstruktur zur Laufzeit zu untersuchen und zu verändern
  - meistens nur in Verbindung mit einer virtuellen Maschine oder einem Interpreter
  - Beispiele: C#, Delphi, Java, Lisp, Perl, Smalltalk, ...
- Skriptsprachen
  - Automatisierung von „kleinen”, häufigen Aufgaben
  - oft zur Steuerung von anderen Programmen
  - Beispiele: sh, bash, VBScrip, AppleScript, BeanShell, JavaScript, Perl, PHP, Ruby, ...
- Synchrone Sprachen
  - optimiert für Echtzeitbetrieb und eingebetteten Systemen

Weitere Kategorien VIII

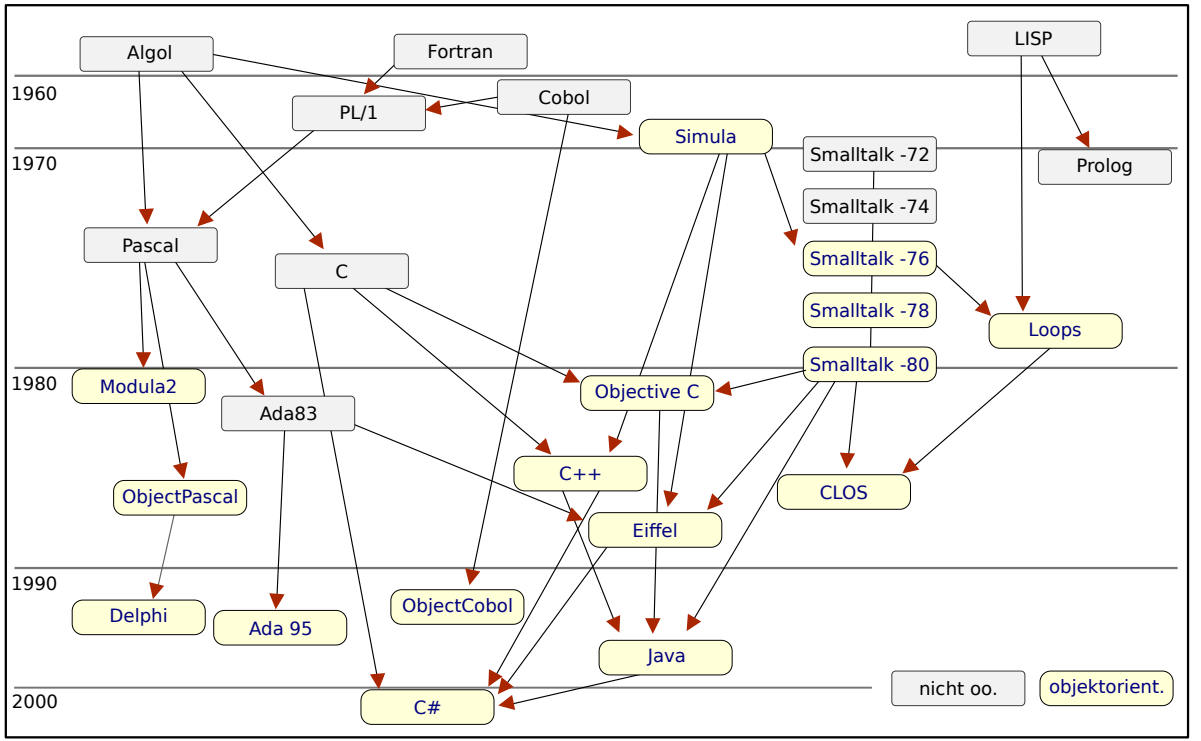
- Beispiele: Averest, Esterel, Lustre

Weitere Kategorien IX

- Visuelle Programmiersprachen
  - Zweidimensionale, grafische Programmierung
  - Beispiele: LabVIEW, Mindscript, VisSim, ...
- XML-basierte Sprachen
  - zur Verarbeitung von XML-Strukturen
  - Beispiele: XPath, XQuery, XSLT, ...
- Auszeichnungssprachen
  - zur Beschreibung von Dingen, sehr häufig Dokumenten
  - teilweise auch Turing-vollständig
  - Beispiele: HTML,XML, LaTeX, PostScript, ...

Das Wichtigste in Kürze

- Ziele von Programmiersprachen
- Beschreibung von Programmiersprachen
- Kategorisierung von Programmiersprachen
  - imperativ
  - objektorientiert
  - funktional
  - logisch


Algol
Fortran
LISP
1960
PL/1
Cobol
Simula
1970
Smalltalk -72
Prolog
Smalltalk -74
Pascal
C
Smalltalk -76
Smalltalk -78
Loops
1980
Modula2
Ada83
Objective C
Smalltalk -80
C++
CLOS
ObjectPascal
Eiffel
1990
Delphi
Ada 95
ObjectCobol
Java
nicht oo.
objektorient.
2000
C#

Inhalt

Syntax

- Struktur einer (Programmier)sprache
  - Welche Zeichen sind erlaubt?
  - Wie dürfen die Zeichen zu gültigen Wörtern zusammengesetzt werden?
  - Wie dürfen Wörter zu gültigen „Sätzen” zusammengesetzt werden?
- Keine Aussage über die Bedeutung der Wörter und Sätze

Beispiel in natürlicher Sprache

- „schnelles ein fährt Auto Thomas.”
  - syntaktisch falscher Satz
  - Wortaufbau im Deutschen nicht eingehalten: *Subjekt Prädikat Objekt*
- „Thomas sitzt ein schnelles Auto.”
  - syntaktisch korrekter Satz
  - keine sinnvolle Bedeutung $\Rightarrow$ Semantik

Beispiel in Programmiersprache

```
sum a = b -;
```

- syntaktisch falsche Anweisung
- Struktur einer Zuweisung nicht eingehalten: Variable links, dann '='-Zeichen, dann Summe aus zwei anderen Variablen und am Ende Semikolon

Beispiel in Programmiersprache

```
sum a = b -;
```

- syntaktisch falsche Anweisung
- Struktur einer Zuweisung nicht eingehalten: Variable links, dann '='-Zeichen, dann Summe aus zwei anderen Variablen und am Ende Semikolon

```
String a = "Hallo";
String b = "KdI";
String s = a - b;
```

- syntaktisch korrekte Anweisungen
- Semantische Bedeutung unklar

Beschreibung der Syntax

- Textuell in langatmigen Erklärungen
  Eine if-Anweisung besteht aus dem Wort „`if`", gefolgt von einer Bedingung, gefolgt von einer öffnenden geschweiften Klammern (`{`), gefolgt von einer oder mehreren Anweisungen, gefolgt von einer schließenden geschweiften Klammern (`}`), gefolgt von einem optionalen else-Teil, der aus dem Wort „`else`" ...
- Mit *kontextfreier* Grammatik

```
<if-Anweisung> ::= if <Bedingung> { <Anweisung> }
                   [else { <Anweisung> }]
```

- Notation meistens aufbauend auf *Backus-Naur-Form* (BNF) oder erweiterter BNF (EBNF)
  - eingeführt von John Backus und Peter Naur zur Beschreibung der Syntax von Algol60

EBNF I

- Rechte Seite beschreibt Struktur der linken Seite
  - Trennung durch `::=`
- Nichterminalsymbole
  - kennzeichnen eine weiter zerlegbare Struktur, z.B. if-Anweisung
  - Schreibweise mit spitzen Klammern, z.B. `<if-Anweisung>`
- Terminalsymbole
  - fest definierte Wörter oder Zeichen, z.B. `class`
  - Schreibweise als normaler Text
- Auswahlmöglichkeiten
  - `<Ziffer> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`
  - Trennung der Möglichkeiten durch `|`

EBNF II

- Optionale Teile
  - `<Var-Def> ::= [<Modifier>] <Typ> <Bezeichner>;`
  - Einschluss in eckige Klammern
- Wiederholung (beliebig viele, auch null)
  - `<Zahl> ::= <Ziffer> { <Ziffer> }`
  - Einschluss in geschweifte Klammern
- Verwendung von Metazeichen als Terminalsymbole
  - `<If> ::= if (<Bedingung>) '{' <Anweisung> '}'`
  - Einschluss in einfache Anführungszeichen

Lexikalische Struktur

- Struktur der „Wörter”, auch *Token* genannt
- Reservierte Wörter
  - `if`, `class`, `public`, ...
- Konstanten oder Literale
  - Zahlen, z.B. 1, 42, -5, 0xff, 045, 1.2e5
  - Zeichenketten, z.B. ”Hallo”, ”Kdl”(Anführungszeichen gehören dazu!)
- Sonderzeichen
  - ;, <=, +, ...
- Bezeichner
  - Student, sum, ganz_langer_variablenname, getCount

Semantik

- Bestimmt, was eine syntaktisch korrekte Struktur bedeutet
- `i = i + 1;`
  - Der Wert der Speicherstelle, auf die `i` verweist, ist nach der Ausführung eins größer.
- Beschreibung der Semantik meistens in natürlicher Sprache
- Semantische Überprüfung nach syntaktischer Überprüfung
  - Sind die Datentypen bei Operationen kompatibel?
  - Hat ein Objekt die gewünschte Methode?
  - Darf die Methode aufgerufen werden?
  - Ist ein Bezeichner (noch) gültig?

Formale Semantik

- Versuch, auch Semantik formal zu beschreiben
- Operationale Semantik
  - Festlegung, wie sich ein Programm auf einer bestimmten bekannten Maschine verhält
  - Maschine kann echter Computer oder abstrakte Maschine sein
  - definierender Interpreter oder Compiler, z.B. FORTRAN, Perl
- Denotationale Semantik
  - Definition der Semantik über (mathematische) Funktionen
  - kommt funktionalen Programmiersprachen entgegen
- Axiomatische Semantik
  - Definition der Auswirkung einer Anweisung über Zusicherungen
  - Vorbedingung und Nachbedingung
  - $\{x = A\}\ \ x = x + 1\ \ \{x = A + 1\}$

Typsystem

- Typisierung von „Objekten” (Variablen, Funktionen, Objekten)
- Einschränkung des „Wertebereichs” von Variablen
- Überprüfung auf korrekte Verwendung zur Vermeidung von Laufzeitfehlern

Bestandteile

- Typen
  - eingebaute, „primitive” Datentypen
  - mittels Typdefinition selbstdefinierte Typen
- Deklaration von Programmelementen mit einem bestimmten Typ
- Regeln, um dem Ergebnis von Ausdrücken einen Typ zuzuweisen
  - `x = 5 + 2;` $\Rightarrow$ `x` ist eine ganze Zahl
  - `x = 5.2 + 2;` $\Rightarrow$ `x` ist eine reelle Zahl
- Regel, um korrekten Typ von Zuweisungen zu prüfen
  - `double f = 2;` $\Rightarrow$ OK
  - `int i = 2.1;` $\Rightarrow$ Fehler
- Optional Methoden, um Typen zur Laufzeit zu ermitteln und zu prüfen

Klassifizierung

- Starke Typisierung $\longleftrightarrow$ schwache Typisierung
  - strenge Restriktion auf erlaubten Operationen
  - `x = "2"+ 3;` nicht erlaubt in starken Typsystemen, aber erlaubt in schwachen Typsystemen
  - schwache Typisierung z.B. in Perl oder PHP
- Dynamische Typisierung $\longleftrightarrow$ statische Typisierung
  - Zuweisung des Typs zur Laufzeit basierend auf zugewiesenen Werten
  - Ermittlung des Typs zur Übersetzungszeit basierend auf Deklaration
- Implizite Typisierung $\longleftrightarrow$ explizite Typisierung
  - automatische Ermittlung des Typs über Typinferenz
  - `x = 5.2 + 2;` $\Rightarrow$ `x` muss reelle Zahl sein

Das Wichtigste in Kürze

- Ziele von Programmiersprachen
- Beschreibung von Programmiersprachen
- Kategorisierung von Programmiersprachen
  - imperativ
  - objektorientiert
  - funktional
  - logisch
- Syntax zur Beschreibung der korrekten Struktur
- Semantik zur Beschreibung der Bedeutung
- Typsystem

Literatur


D. J. Barnes und M. Kölling.
Objektorientierte Programmierung mit Java — Kapitel 1–3.
Pearson Studium, 2003, ISBN 978-3-8273-7073-6.


A. Poetzsch-Heffter.
Konzepte objektorientierter Programmierung — Kapitel 1–2.
Springer-Verlag Berlin Heidelberg, 2000, ISBN 3-540-66793-8.


H. Herold, B. Lurz, und J. Wohlrab.
Grundlagen der Informatik — Kapitel 7.6.
Pearson Studium, 2007, ISBN 978-3-8273-7305-2.


Heinz Peter Gumm und Manfred Sommer.
Einführung in die Informatik — Kapitel 2.1, 2.6.
Oldenburg Verlag, 7. Ausgabe, 2006, ISBN 978-3-486-58115-7.


Kenneth C. Louden.
Programmiersprachen (deutsche Übersetzung)
International Thomson Publishing, 1994, ISBN 3-929821-03-6