

Eprog 1. Semester

EBNF Allgemein

- Programm: Folge von Anweisungen, die von einem Computer ausgeführt werden (können)
- Mögliche Anweisungen: Programmiersprache
- EBNF:
 - Extended
 - Backus (backup)
 - Naur (normal)
 - Form
- Beschreibt die Syntax einer Sprache

Wieso EBNF in Eprog

- Praktische Beweise relevanter Eigenschaften
- Jede EBNF Beschreibung hat zwei Seiten

EBNF

- Vier Elemente (control forms) die in Java wiedergefunden werden
 - Aufreihung, sequence
 - Entscheidung, decision (Auswahl und Option)
 - Wiederholung, repetition
 - Rekursion, recursion
- Statt kursiv zwei Klammern <>

EBNF Regel

RHS:

- Genaue Beschreibung für den Namen
- Kann enthalten:
 - Namen (von EBNF Regeln)
 - Buchstaben
 - Kombinationen der vier Kontrollelemente

LHS:

- Ein Wort, kursiv, kleingeschriebe

Pfeil:

←

Symbol	Bedeutung	Beispiel
Regelname	Bezeichnet T- oder NT-Symbol	Zuweisung
„...“	Anführungszeichen für Strings	„2“ oder „Programm“
<=	Trennt die Regelseiten	<digZiffer> <= 0 1;
;	Schliesst Regel ab	<grossbuchstabe> <= A Z;
[]	Fasst Optionen zusammen	<zahl> <= [-] {digZiffer},
()	Gruppierung von Symbolen	Buchstabe, (Buchstabe Ziffer)
{}	0, 1, beliebige Wiederholungen	<positiveZahl> <= {ziffer, ziffer}

- <vorzeichen> <= [+|-] → möglich auch nichts zu wählen → dann Epsilon
- 0 Wiederholungen heisst – fehlt
- Konvention: Reihenfolge der Regeln und gewählte Namen wichtig:
 - Von einfach nach komplex, relevante Namen
 - Name der letzten Regel ist der Name der relevanten Beschreibung

Tabellen

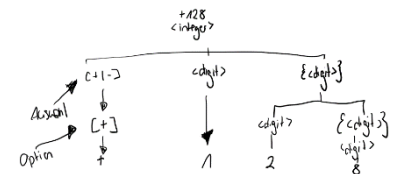
Jede Zeile wird aus Vorgängerzeile durch Regel abgeleitet

1. Ersetze Name durch entsprechende Definition
2. Wahl Alternative
3. Entscheidung ob optionales Element dabei ist oder nicht
4. Bestimmung der Zahl der Wiederholung

$+128$
 $\langle \text{integer} \rangle$
 $[+|-] \langle \text{digit} \rangle^* \langle \text{digit} \rangle^+$
 $[+|-] \langle \text{digit} \rangle \{ \langle \text{digit} \rangle^* \}$
 $+ \langle \text{digit} \rangle \{ \langle \text{digit} \rangle^* \}$
 $+1 \langle \text{digit} \rangle \{ \langle \text{digit} \rangle^* \}$ *2x Wiederholung*
 $+12 \langle \text{digit} \rangle^+$
 $+128$

Ableitungsbäume

- Graphische Darstellung Beweise durch Tabelle
- Oben: Name EBNF Regel, mit der das Symbol übereinstimmen soll
- Unten: Symbol
- Kanten zeigen welche Regeln es uns erlauben von einer Zeile zur nächsten (in der Tabelle) zu gehen



Allgemeines

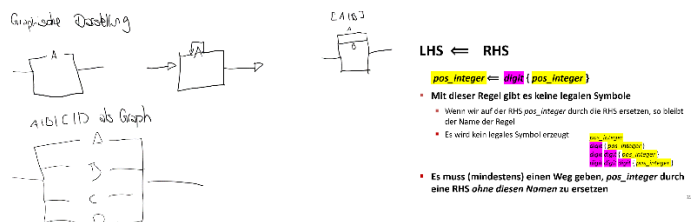
- „{„ als Symbol per se → in eckigen Rahmen
- Jede EBNF Beschreibung definiert eine Sprache: Menge der legalen Symbole
- Äquivalente EBNF Beschreibungen erkennen dieselben legalen und illegalen Symbole

Syntax und Semantik

- Syntax: Form (legt nur die Form fest) → EBNF
- Semantik: Bedeutung

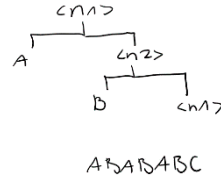
Bedeutung von Mengen

- Mehrfach Nennungen sind nicht wichtig: $\{1, 2, 3, 2, 2, 2, 3\}$ äquivalent zu $\{1, 2, 3\}$
- Reihenfolge nicht wichtig: $\{1, 2, 3\}$ äquivalent zu $\{3, 2, 1\}$
- Kanonische (in Übereinstimmung mit Regel) Darstellung: geordnet von kleinster [links] nach grösster Zahl [rechts]
 - Kann nicht durch EBNF erzwungen werden



Rekursion

- Muss (mindestens) einen Weg geben, Rekursion durch eine RHS ohne diesen Namen zu ersetzen (|)
- Direkt Rekursiv: Name in der Definition wird verwendet
- Wiederholung zu Rekursion
- Nicht alle Rekursion zu Wiederholungen
- Indirekte Rekursion:
 - $\langle \text{name1} \rangle \Rightarrow A \text{ name2};$
 - $\langle \text{name2} \rangle \Rightarrow B \text{ name1} \mid C$



EBNF Geschichte

- BNF: nur Rekursion und Auswahl
- Daher „E“ für Extended

Eprog Einfache Java Programme

Java Programme

- Ganzes Programm
 - Braucht Compiler
- Für jede Datei...

Erstellen / übersetzen (compilieren)

- - Ausführen
 - Modifizieren
- Einzelne Anweisungen
 - Braucht Shell
- Für jede Anweisung...
 - Read
 - Evaluate
 - Print
 - Loop
 - REPL

- Name des Programms gleich Name der Datei
- Viele (Textbearbeitungs)Methoden lassen das Objekt mit dem die Methode arbeitet unverändert
- Bezeichner: Muss mit Buchstaben (gross/klein) anfangen (oder _ \$)

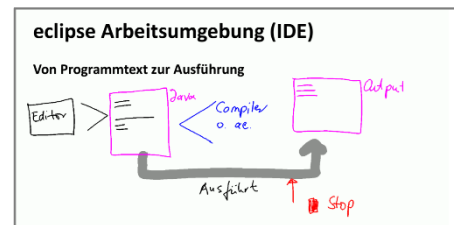
Sonderzeichen

- \t tab character
- \n neue Zeile (new line character)
- \” double quote character
- \\ backslash character

Methoden

- Strukturieren die Anweisungen
- Wiederholungen zu vermeiden
- Main wird automatisch ausgeführt
- Aufrufen:
 - Object.methodName();
 - Ohne Objekt → mit static
- Abfolge der Ausführung von Anweisungen: Kontrollfluss (control flow)
- Java: Anweisungsreihenfolge ist explizit

Infrastruktur – Java Programme



12

Zusammenfassung

```
public class name {  
    statement;  
    statement;  
    ...  
    statement;  
}
```

class: ein Programm mit Namen name
method: Gruppe von Anweisungen mit Namen main
statement: Anweisung die ausgeführt werden soll

15

Methoden

```
public class name {  
    public static void main(String[] args) {  
        statement;  
        ...  
    }  
    public static void helper() {  
        statement;  
        ...  
    }  
}
```

method: Gruppe von Anweisungen mit Namen main
method: Gruppe von Anweisungen mit Namen helper

16

Typen/Variablen

- Typ beschreibt Eigenschaften von Daten
 - Wertebereich
 - Operationen
 - Darstellung (welche Folge von 0 und 1 für einen Wert gewählt wird)
- Typen verhindern Fehler, erlauben Optimierung
- Beschreibt Menge (Kategorie) von Daten Werten
- Variable: benötigt Name und auf was für Werte sich die Variable beziehen kann

Modulo

- Finde letzte Ziffer einer ganzen Zahl
- Finde letzte 4 Ziffern
- Entscheide ob Zahl gerade ist
- Rangordnung (Precedence) ist wichtig: * / stärker als + -

Assoziativität («Associativity») -- Bindung

- Die Assoziativität eines Operators \odot hält fest wie ein Operand zu verwenden ist:

$$X \odot Y \odot Z$$

- Y ist mit dem *linken* Operator verknüpft: links-assoziativ («left-associative»)

$$X \odot Y \odot Z = (X \odot Y) \odot Z$$

- Y ist mit dem *rechten* Operator verknüpft: rechts-assoziativ («right-associative») $X \odot Y \odot Z = X \odot (Y \odot Z)$

Assoziativität

- Links-assoziativ: Y ist mit dem *linken* Operator verknüpft («left-associative», «left-to-right associative»)

$$X \odot Y \odot Z = (X \odot Y) \odot Z$$

Viele der uns bekannten Operatoren: +, *,

- rechts-assoziativ: Y ist mit dem *rechten* Operator verknüpft («right-associative», «right-to-left associative»)

Später werden wir Beispiele sehen (es gibt einige!)

- Es gibt Operatoren die sind *assoziativ* (in der Mathematik)

- Rechts—assoziativ und links—assoziativ: $(X \odot Y) \odot Z = X \odot (Y \odot Z)$

Typ Umwandlungen

- Explizite Umwandlungen heissen cast, type cast
- (type) expression
 - (int) ((double) 19/5)
- type ist Operator → rechts-assoziativ

Variable

Name, der es erlaubt, auf einen gespeicherten Wert zuzugreifen

- Deklaration
- Initialisierung
- Gebrauch
- Zuweisung ist keine algebraische Gleichung!

Operanden und Operatoren

- Operand wird vom Operator mit höherer Rang Ordnung («precedence», «Präzedenz») verwendet
- Wenn zwei Operatoren die selbe Rang Ordnung haben, dann entscheidet die Assoziativität
- Wenn zwei Operatoren die selbe Rang Ordnung und Assoziativität haben, dann werden die (Teil)Ausdrücke von links nach rechts ausgewertet.
- Wenn etwas anderes gewünscht wird: Klammern verwenden!

Deklaration

Variable müssen deklariert sein bevor sie verwendet werden können.

type name;

BNF Description *variabledeclaration*

typeidentifier \Leftarrow *bezeichner*
variableidentifier \Leftarrow *bezeichner*
variabledeclaration \Leftarrow *typeidentifier variableidentifier* { , *variableidentifier* } ;

24

Ableiten mit Aussagen

- Positionen im Code haben Namen (Annahme)
 - Point A
 - Point B
- Alle Anweisungen die davor erscheinen, sind ausgeführt, wenn wir diesen Punkt erreichen
- Keine Anweisung danach wurde ausgeführt
- Hoare Logik:
 - Vorwärts und rückwärts schliessen
 - Von einer Anweisung zu mehreren Anweisungen und Blöcken
- Wichtig für die Definition von Schnittstellen (zwischen Modulen) wenn wir entscheiden müssen welche Bedingungen erfüllt sein müssen (um eine Methode aufzurufen).
- Vorwärts schliessen:
 - Simuliert die Ausführung des Programms (für viele «Inputs» «gleichzeitig»)

Beispiel

- **Vorwärts schliessen**
 - Vom Zustand vor der Ausführung eines Programms(segments)
 - Nehmen wir an wir wissen (oder vermuten) $w > 0$

```
// w > 0
x = 17;
// w > 0  ∧  x == 17
y = 42;
// w > 0  ∧  x == 17  ∧  y == 42
z = w + x + y;
// w > 0  ∧  x == 17  ∧  y == 42  ∧  z > 59
```
 - Jetzt wissen wir einiges mehr über das Programm, u.a. $z > 59$

Beispiel

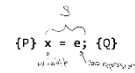
- **Rückwärts schliessen:**
 - Nehmen wir an wir wollen dass z nach Ausführung negativ ist

```
// w + 17 + 42 < 0
x = 17;
// w + x + 42 < 0
y = 42;
// w + x + y < 0
z = w + x + y;
// z < 0
```

54

- Bestimmt was sich aus den ursprünglichen Annahmen herleiten lässt
- Sehr praktisch wenn eine Invariante gelten soll (Invariant = etwas, das sich nicht ändert)
- Rückwärts schliessen
 - Bestimmt hinreichende Bedingungen

Zuweisungen



- Bilden wir Q' in dem wir in Q die Variable x durch e ersetzen
- Das Tripel ist gültig wenn:
Für alle Zustände des Programms ist Q' wahr wenn P wahr ist
- D.h., aus P folgt Q' , geschrieben $P \Rightarrow Q'$

Beispiel

```
{z > 34}
y = z+1;
{y > 1}
```

Q' ist $\{z+1 > 1\}$

Folgen von Anweisungen

- Einfachste Folge: zwei Statements
 $\{P\} S1; S2 \{Q\}$
- Tripel ist gültig wenn (und nur wenn) es eine Aussage R gibt so dass
 1. $\{P\} S1 \{R\}$ ist gültig und
 2. $\{R\} S2 \{Q\}$ ist gültig.

Pre- und Postconditions

- Precondition: notwendige Vorbedingung, die erfüllt sein müssen (vor Ausführung einer Anweisung)
- Postcondition: Ergebnis der Ausführung (wenn Precondition erfüllt)
- Precondition, Anweisung und Postcondition hängen zusammen
- Aussagen (Pre, Post) sind logische (bool'sche) Ausdrücke die sich auf den Zustand eines Programms beziehen
- Zwischen $\{$ und $\}$ steht eine logische Aussage

Hoare Tripel

- Zwei Aussagen und ein Programmsegment
 - $\{P\} S \{Q\}$
 - P Precondition
 - S Statement
 - Q Postcondition
 - Gültig:
 - Zustand P gültig, Ausführung von S gibt Zustand Q
 - Wenn P wahr ist vor der Ausführung von S , dann muss Q nachher wahr sein

Beispiel

- Alle Variable sind int, kein Overflow/Underflow
 $\{z >= 1\}$
 $y = z+1;$
 $\{y > 1\}$
 $w = y*y;$
 $\{w > y\}$
- Sei R die Aussage $\{y > 1\}$
 1. Wir zeigen dass $\{z >= 1\} \Rightarrow \{y > 1\}$ gültig ist.
Regel für Zuweisungen: $z := z+1 \Rightarrow y > 1$ impliziert $z+1 > 1$
 2. Wir zeigen dass $\{y > 1\} \Rightarrow \{w > y\}$ gültig ist.
Regel für Zuweisungen: $y := y*y \Rightarrow y > 1$ impliziert $y*y > y$

Boolesche Operatoren

- Ausdrücke mit Vergleichsoperatoren können durch boolesche Operatoren verknüpft werden

Operator	Bedeutung	Beispiel	Wert
&&	and	$(2 == 3) \&\& (-1 < 5)$	false
	or	$(2 == 4) (-1 < 5)$	true
!	not	$!(2 == 3)$	true

- «Wahrheitstabelle» für diese Operatoren, für Aussagen p und q :

p	q	$p \&\& q$	$p q$
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Verzweigungen (if-Anweisungen)

- Nur manche Anweisungen ausführen
- Bedingter Ausführung
- If Anweisung:
 - Ist Test wahr?
 - Ja: ausführen
 - Nein: folgende Anweisung ausführen
- If-else-Anweisung
 - Ist Test wahr?
 - Ja: if ausführen
 - Nein: else ausführen
 - Danach folgende Anweisung ausführen
- If-else-if-Anweisung
 - Ist Test 1 wahr?
 - Ja: Test ausführen

Verschachtelte if-Konstrukte

- Gehen ein 2-Flad mit Anweisungen (geprägter Konstruktor)
- Gehen ein 1-Flad mit Anweisungen (geprägter Konstruktor)
- Gehen ein 1-Flad mit Anweisungen (geprägter Konstruktor)
- Gehen ein 1-Flad mit Anweisungen (geprägter Konstruktor)

If-Statements

$\{P\} \text{ if } (b) \ S1 \text{ else } S2 \{Q\}$

- Tripel ist gültig wenn (und nur wenn) es Aussagen $Q1, Q2$ gibt so dass
 1. $\{P \wedge b\} S1 \{Q1\}$ ist gültig und
 2. $\{P \wedge !b\} S2 \{Q2\}$ ist gültig und
 3. Nach dem if-Statement gilt Q , d.h.
 $(Q1 \Rightarrow Q) \wedge (Q2 \Rightarrow Q)$

- Nein: ist Test 2 wahr
- Ja..

•

Wenn $P1 \rightarrow P2$ (also $P1$ impliziert $P2$) gilt dann sagen wir:

- $P1$ ist stärker als $P2$
- $P2$ ist schwächer als $P1$
 - Schwächste heisst: hat die wenigsten Annahmen/Einschränkungen so dass Q gilt
 - Jede Precondition P so dass $\{P\} S \{Q\}$ gültig ist, ist dann stärker als $P \gg$, d.h. $P \rightarrow P \gg$

Bedingte («short-circuit») Auswertung

- Reihenfolge von Statements beachten
 - Wenn Bruch nicht eine Null unten haben darf:
 - $B \neq 0$ vor $(a/b > 0)$

Schleifen

- Bestimmte Schleifen (definite loop) Anzahl der Ausführungen des Rumpfes ist vor Beginn der Ausführung der Schleife bekannt
- Unbestimmte Schleifen: Anzahl Iterationen ist nicht vorher bekannt
 - While loop

Methode

```
Public static void main (String[] args) {
    Vorsatz(); //Aufrufer, caller
}
```

```
Public static void vorsatz() { //Aufgerufene, callee
}
```

- Parameter: Wert den eine aufgerufene Methode von der aufrufenden Methode erhält
 - Bei Deklaration von Methode, geben wir an, dass Methode einen Parameter braucht
 - Bei Aufruf der Methode, wir geben Wert für Parameter an
 - Parameter in Deklaration einer Methode heisst formaler Parameter
 - Der übergebene Wert heisst tatsächlicher Parameter (Argument)
- Übergabe von Parameter:
 - Bei Aufruf
 - Wert in Parameter Variable gespeichert
 - Anweisungen der Methode werden ausgeführt (mit diesem Wert für Parameter Variable)
- Übergabe von Werten (value semantics)
 - Wenn aktueller Parameter durch Variable V bestimmt wird, dann wird Wert dieser Variable vom Aufrufer kopiert (value semantics)
 - Veränderungen der Parameter Variable (formaler Parameter) in der aufgerufenen Methode haben keine Auswirkung auf V
- Scanner
 - Scanner in Bibliothek `java.util` definiert
 - `Scanner name = new Scanner (System.in)` → Kreiere neues Objekt
 - `int alter = myConsole.nextInt();`
 - Eingabe wird mit Enter (return) Taste abgeschlossen
 - Folge von Zeichen die der Scanner liest: Token
 - Nach Zwischenraum getrennt

Warum ist das interessant?

- Stellen wir uns vor:
 - Es gilt $\{P\} S \{Q\}$, und
 - P ist schwächer als eine Aussage $P1$, und
 - Q ist stärker als eine Aussage $Q1$

- Dann gilt:
 - $\{P1\} S \{Q\}$
 - $\{P\} S \{Q1\}$
 - $\{P1\} S \{Q1\}$

37

Beispiel

Alle Variable sind int, kein Overflow/Underflow

```
{true}
if (x > 7) { y = x; }
else { y = 20; }
{y > 5}
```

- Sei $Q1 \{y > 7\}$ (andere Aussagen gehen evtl. auch)
- Sei $Q2 \{y == 20\}$ (andere Aussagen gehen evtl. auch)

- Mit der Regel für Zuweisungen können wir zeigen $\{true \wedge x > 7\} \{y = x; \{y > 7\}\}$
- Mit der Regel für Zuweisungen $\{true \wedge x \leq 7\} \{y = 20; \{y == 20\}\}$
- Dann zeige dass $(y > 7) \vee (y == 20) \Rightarrow y > 5$

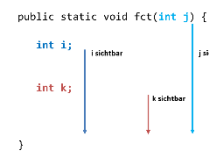
38

Sichtbarkeit von Variablennamen

- Scope: Bereich in dem Name sichtbar ist
 - Teil eines Programm wo Variable sichtbar ist
 - Variable müssen deklariert sein bevor sie sichtbar sind
 - Eindeutige Deklaration
 - Variable → kann nicht «unsichtbar» werden
- Java: Methoden können nicht in anderen Methoden geschachtelt sein → Aufruf ok, keine eigene Deklaration
- Variable kann in einem Sichtbarkeitsbereich nicht mehrmals deklariert werden
- Regeln für:
 - Lesbarkeit Programm
 - Vereinfachung Verwaltung Speichers
 - Platz für Variable eines Basistypes muss nur in dem Block organisiert werden, in dem die Variable deklariert ist
 - Werte die in Variable eines Basistypes gespeichert werden, verschwinden am Ende des Blockes

{ und } strukturieren ein Programm

```
public static void fct(int j) {  
    int i;  
    int k;  
}
```



Strings

- Nur eine Zeile → \n → newLine
- Java Typ String, definiert in Standard Bibliothek
- «+» erzwingt Konversion von anderen Typen zu String
- Strings sind keine Arrays
- «hello».substring(0,1) «h» → kein Char
- Wie viel Speicher benötigt ein String?
- == nur Basistypen
- Next() → Methode liest ein Wort (d.h. keine Zwischenräume) als String
- Keine Anführungszeichen

String Methoden die String liefern

Method name	Description
substring(index1, index2)	the characters in this string from index1 (inclusive) to index2 (exclusive)
substring(index1)	if index2 is omitted, grabs till end of string
toLowerCase()	a new string with all lowercase letters
toUpperCase()	a new string with all uppercase letters
trim()	a new string whose value is this string, with all leading white spaces removed.
trimTrailing()	a new string whose value is this string, with all trailing white spaces removed.

▪ «white space» -- Leerzeichen (blank, space), Tabulatorzeichen, LineFeed/CarriageReturn/Return/Enter/Zeilenumbruch ...

Inkrement und Dekrement

Variable wird verwendet und dann verändert! Dies gilt auch in Ausdrücken

Bedingte Auswertung und Kurzformen

- && || links assoziativ
- && stoppt sobald Teilasdruck false ist
- || stoppt sobald Teilausdruck true ist

Loops

- Um-Eins-Daneben-Fehler
- Schleife wurde einmal zuviel/zuwenig durchlaufen

Do-While: Ausführung im Loop ausführen, ist Test wahr → ja / nein → Anweisung nach Loop ausführen oder wiederholen

Invariante?

Weitere Kurzformen – manchmal nützlich

▪ x++ und j-- heissen Post-Increment bzw. Post-Decrement Operator, da die Veränderung (von x und j) gemacht wird *nachdem* der Wert (von x oder j) gelesen («gebraucht») wurde.

▪ Es gibt auch Operatoren, die die Veränderung (Increment oder Decrement) durchführen *bevor* der Wert gelesen wurde; dies sind der Pre-Increment bzw. Pre-Decrement Operator: ++j oder --j.

Beispiele

```
int x = 2;  
System.out.println(++x); // x = x + 1; x now stores 3  
System.out.println(++x); // x = x + 1; x now stores 4
```

Output:
3
4

87

Kapitel 3.0 Arrays

- Array → Referenzvariable
 - Genauer: Variable of reference type
- `Int[] myArray;` → beliebige Länge
- Referenzvariable erlaubt Zugriff auf ein Array, ein Objekt
- Zeiger verweisen auf Arrays
- `A = b` → rechte Seite einer Zuweisung zu einer Referenzvariable muss auch Referenzvariable sein
 - Typ der Elemente muss übereinstimmen
 - Anzahl Elemente muss NICHT übereinstimmen
- `A = null;`
 - Null heisst, dass a auf kein Array verweist
- **Reference Semantics:**
 - `X=y;` Array wird nicht kopiert sondern beide Variablen beziehen sich nun (verweisen) auf den selben Array;
 - Aliasing: zwei oder mehr Referenzvariable verweisen auf den selben Array
 - Warum?:
 - Effizienz → Kopieren grosser Arrays kostet viel Zeit
 - Programmstruktur → Sinnvoll wenn verschiedene Methoden mit gemeinsamen Array arbeiten
 - Objekte verwenden auch Reference Semantics

Kapitel 4.0 Klassen

- Klassen: verschiedene Zwecke:
 - Implementation eines Algorithmus oder Application, bietet Service an
 - Implementation verschiedener Dienste einer Java Bibliothek
 - Von uns entwickelte Dienste
- Klasse enthält Programm
- Ruft automatisch «main» auf
- Zustand plus Operationen: Typ
 - (Daten) Type beschreib/bestimmt zulässige Operationen und zulässige Werte
- Klassen beschreiben einen Typ
 - Typ beschreibt Eigenschaften von Daten
 - Typ beschreibt Menge von Daten Werten
 - Basistypen beschreiben Typ ohne Klasse
- Objekt
 - Sammelbegriff für alle Datenwerte, die durch eine Klasse beschrieben werden
 - Objektexemplar
 - Muss zuerst erschaffen werden
 - Als Parameter: wird eine Referenzvariable übergeben
 - Programm Einheit mit der wir
 - Klasse: Neue Art/Typ von Objekten beschreiben können → beschreibt Form/Funktionalität von Objekten
 - Objekt: Ein Gebilde das Zustands und Verhalten verbindet

- Abstraktion: reduzierte Beschreibung, lässt irrelevante Details weg
- Klasse: Vorlage (Mustervorlage, Schablone) die Objekte beschreibt
- Attribut: Variable innerhalb eines Objektes die Teil des Objekt Zustandes ist
 - Jedes Objekt hat seine eigene Kopie jedes Attributes
 - Wir sagen «die Referenzvariable wird dereferenziert» um auf ein Attribut zuzugreifen
 -