VORLESUNG

Automatisierung industrieller Workflows

Teil C: Die Sprache SLX

- Einführung und Basissprache -

Joachim Fischer

Position

- **Teil A**
  Aspekte von Modellierung und Simulation dynamischer Systeme
- **Teil B**
  Die Modellierungssprache UML
- **Teil C**
  Die ausführbare Modellierungssprache SLX
- **Teil D**
  Modellierung von Lieferketten

- **C.1**
  Einführung und Basissprache
- **C.2**
  Stochastische Prozesse in SLX
- **C.3**
  Vertiefung der SLX-Basissprache
- **C.4**
  GPSS-Elemente
- **C.5**
  DISCO-Elemente
- **C.6**
  Basissprache (Ergänzung)

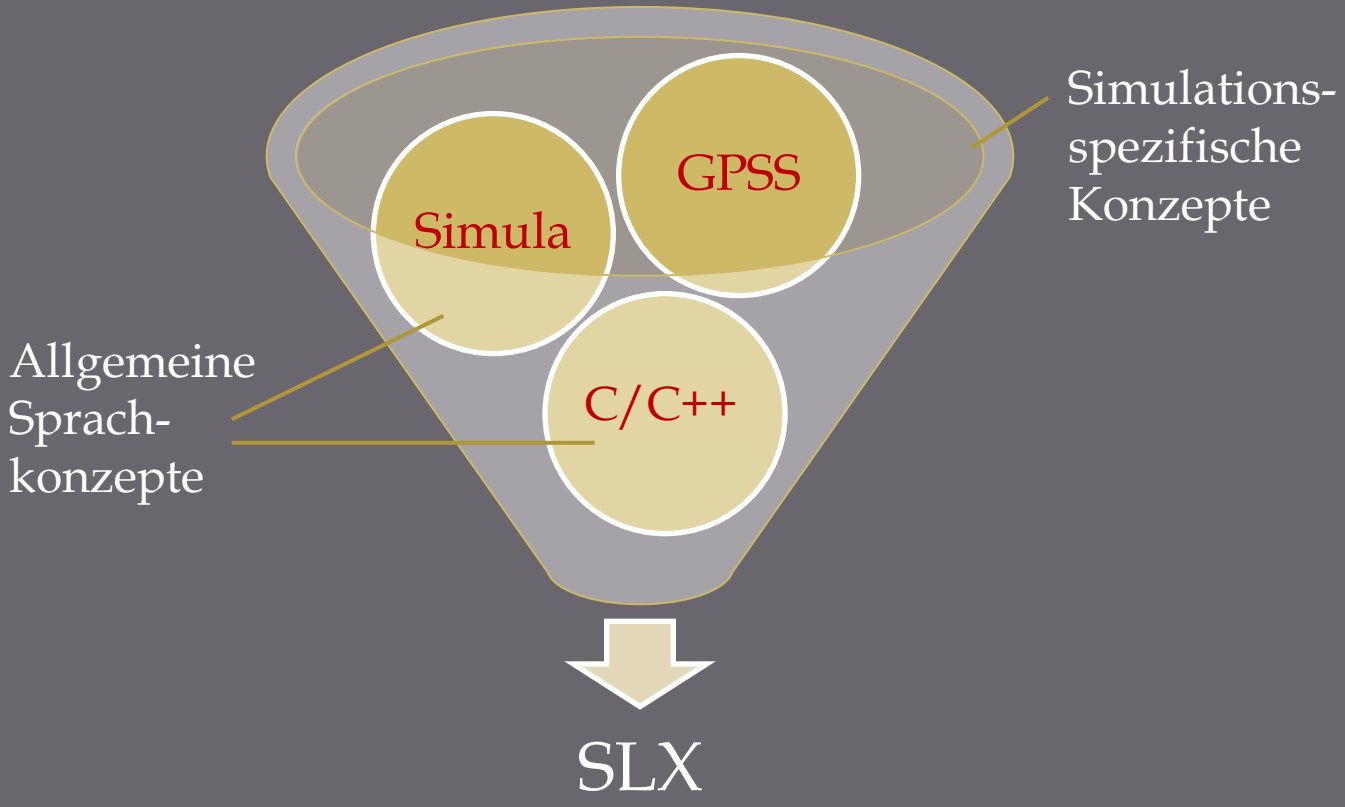
Position

- **Teil A**
  Aspekte
  dynamis
- **Teil B**
  Die Mod
- **Teil C**
  Die ausf
  SLX
- **Teil D**
  Modellie

- **C.1**
  Einführung und Ba
- **C.2**
  Stochastische Pro
- **C.3**
  Vertiefung der SL
- **C.4**
  GPSS-Elemente
- **C.5**
  DISCO-Elemente
- **C.6**
  Basissprache (Erg

1. SLX-Überblick
2. Syntaxkonventionen
3. Elementare Datentypen
4. Klassen
5. Objekte
6. Typ
7. Set
8. Prozeduren
9. Einfache Ausgabe
10. Modellierungselemente in SLX (allgemein)
11. Prozesse und Pucks
12. SLX- Laufzeitsystem (Simulator-Kern)
13. Zeitkonzepte
14. Behandlung von Zeit- und Zustandsereignissen

SLX-Überblick



SLX-Überblick

SLX

- bedeutet Simulation Language with Extensibility 2.0
- hat Sprachumfang entspr. Teilmenge von C/C++ und
  Simula (Koroutinen)
  enthält auch GPSS-Konzepte
  als Paket (Bibliothek)
- verzichtet auf komplizierte Sprachkonstrukte
  (z.B. Zeigerarithmetik,
  Zeiger auf Zeiger)
- einige Containertypen aus STL)
- fügt neue **Schlüsselwörter** als
  Simulationskonstrukte hinzu
  (z.B. advance, wait until, forever)
- **Class** verkörpert als Standardfall
  aktive Klassen

```
pointer(X) x = new X();
x->i = 2;
```

```
set(X) collection_of_x;
place x into collection_of_x;
```

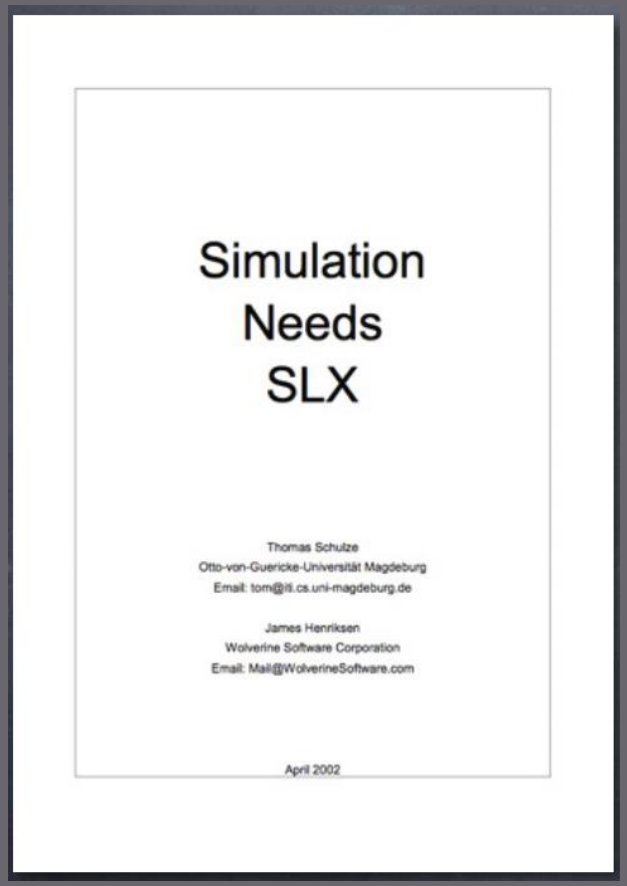
```
class X {
   control int i;

    actions {
                ...
                wait until i == 2;
                advance 10;
    }
}
```

SLX-Überblick

- Version 1.0 >> objektbasiert
  - keine Vererbung
  - nur Objekt-Komposition möglich
- Version 2.0 >> objektorientiert
  - Einfachvererbung zwischen Klassen,
  - Mehrfachvererbung zwischen Interfaces
  - (ähnlich wie in Java)
- Benutzung der Version 2.0 muss explizit eingeschaltet werden:

  #define SLX2 ON

SLX-Überblick



Literatur:

- „Simulation Needs SLX“ (SLX-Handbuch in deutscherSprache)

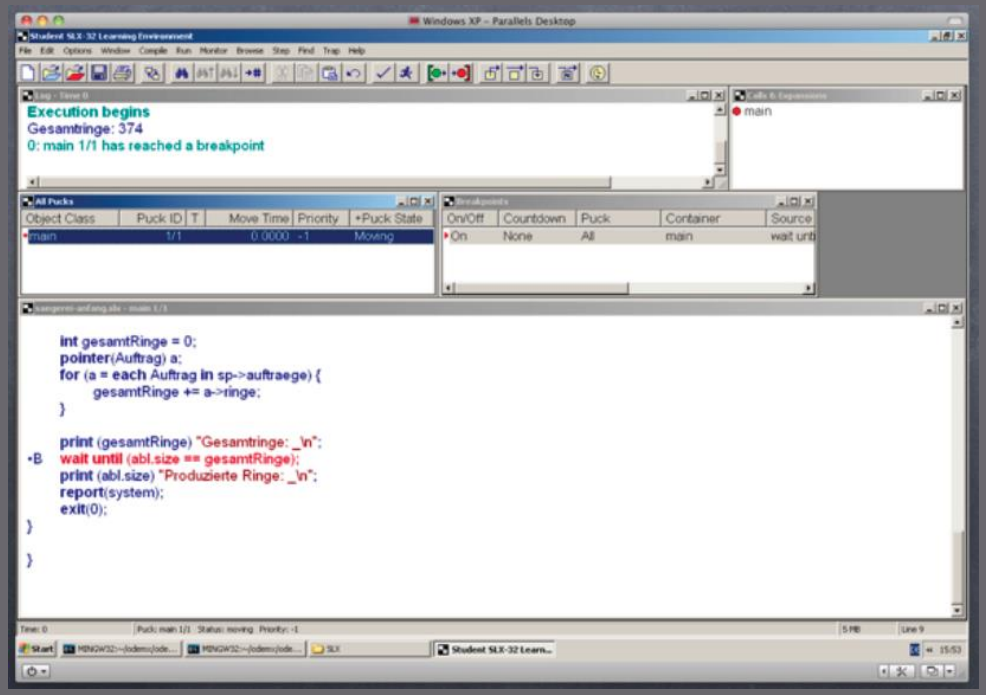
- SLX von Wolverine Software Corporation (USA) ab Mitte 90er Jahre entwickelt
- SLX besteht aus besteht aus der Sprache SLX und einer Entwicklungsumgebung
- Die SLX Sprache ist eine universelle objektbasiert Sprache zur Modellierung und Simulation diskreter Prozesse
- Als objektbasierte Sprache (SLX 1.x) verfügt sie nicht über alle typischen Merkmale einer objektorientierten Sprache, SLX 2.x ist objektorientiert

http://isgwww.cs.uni-magdeburg.de/pelo/sa/SimulationNeedsSLX.pdf

SLX-Überblick

Windows-basierte IDE mit Editor & Simulator/Debugger

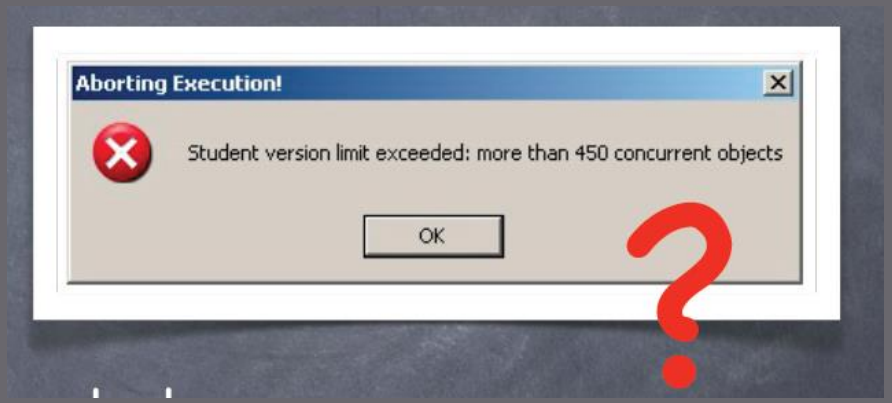
http://www.wolverinesoftware.com/



- kostenfreie Studentenversion
- Beschaffung einer kommerziellen Version ist erfolgt

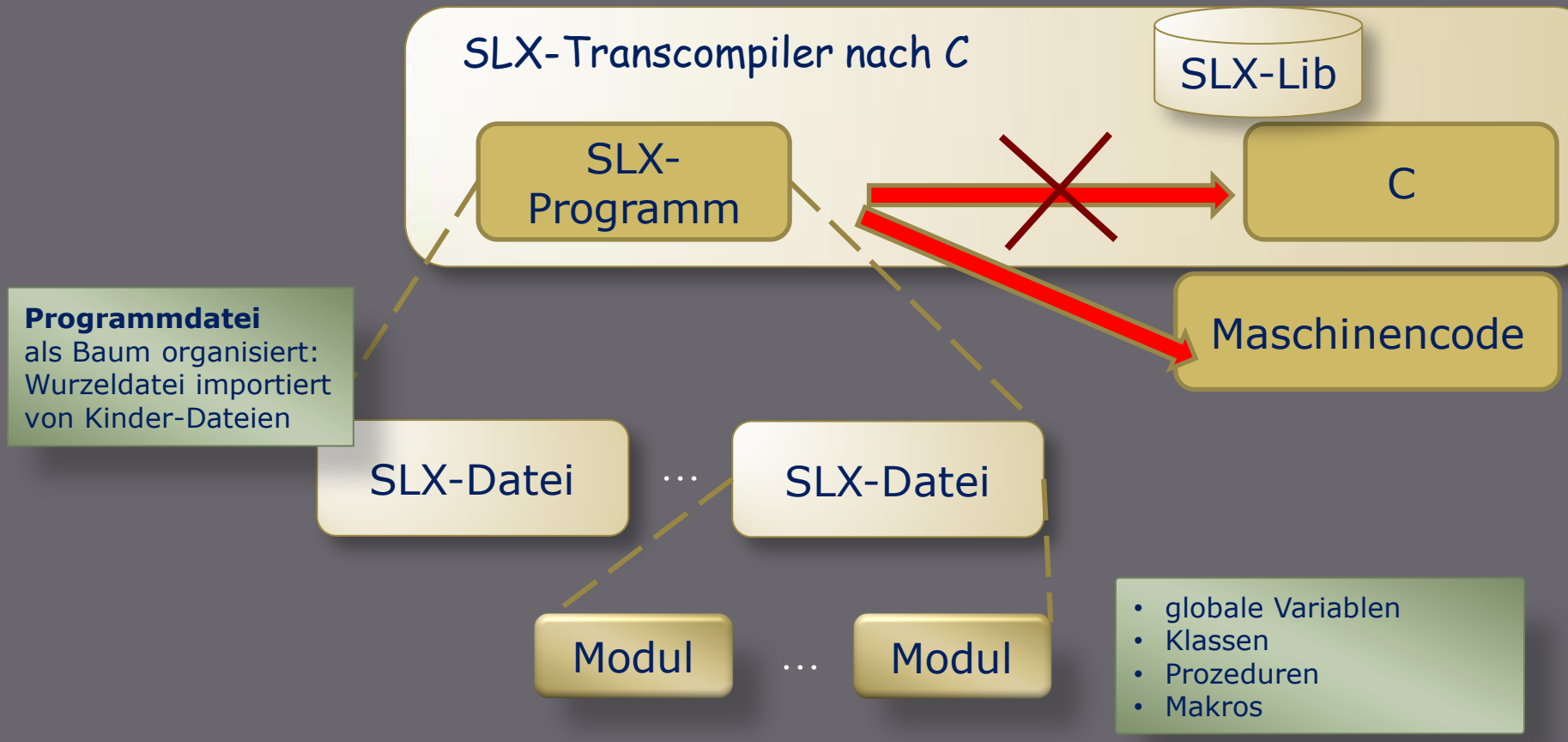
SLX-Überblick

- Studentenversion ist eingeschränkt



  450 Objekte
- (gleichzeitig aktiv -egal ob aktive oder passive Objekte)
- und max. 500 Pucks
  (für jeden Prozess ex. mind. 1 Puck)

SLX-Überblick



genau ein Modul enthält das Hauptprogramm
procedure main()

```
module SimpleModule {
    procedure main() {
        print "Hello SLX World!\n";
    }
}
```

SLX-Überblick

Klasse

- **Prototypbeschreibung** für die Instanzen der Klasse
- definiert **Attribute** ihrer Instanzen
- definiert **Operationen** über Instanzen (Methoden)
- besitzt **Properties** (ausgezeichnete Operationen mit Nutzercode)
- kann parametrisiert werden (mit impliziten Standardkonstruktor)

Ship
ship_kind: Ship_Kind tonnage: **float**
procedure op1 (…) {…} **procedure** op2 (…) {…}
report {…} …

«enum» Ship_Kind
freight , passenger , yacht

elementare Datentypen:
int, **float**, …, **pointer**

Felder von elementaren DT

SLX-Überblick

Stärken von SLX aus Sicht der Entwickler

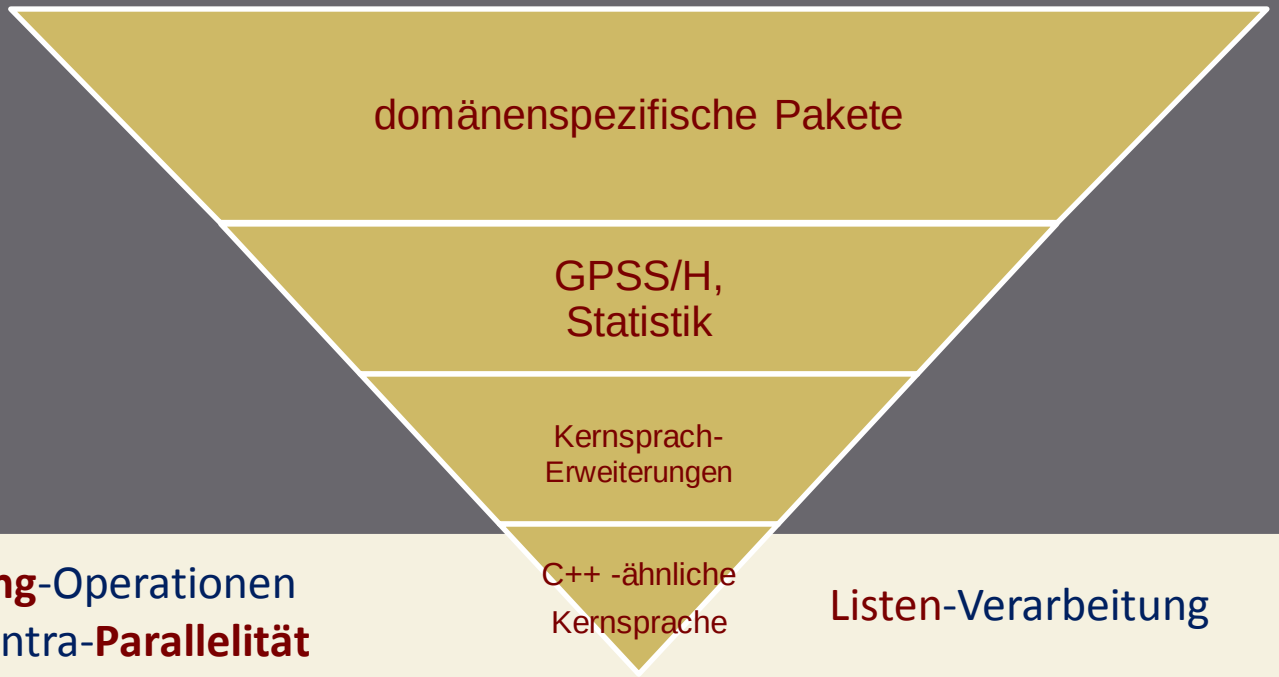
- wichtigste Aspekte sind
  Effizienz und **Erweiterbarkeit** der Sprache

  SLX bietet sowohl eine
  1. Sammlung wohl-definierter und effizient implementierter **Kernsimulations-Primitiven**
  2. als auch **Mechanismen zur Erweiterung** der Kernprimitiven für vielfältige domänspezische Applikationen

- SLX's verbindet **Ausdrucksstärke** mit der Fähigkeit zu einer schnellen und klaren **Fehleranalyse** bei
  - Modellkonstruktion (Compile-Zeit) und
  - Modellausführung (Laufzeit).

im Vergleich zu C++, Java, C#

Hierarchie von SLX-Sprachkonzepten



vordefinierte Module

Scheduling-Operationen
Inter- u. Intra-**Parallelität**

Listen-Verarbeitung

control-Variablen
wait until-Konstrukt

vereinfachtes Pointer-Konzept
(ohne Arithmetik,
Smart-Pointer (ähnlich zu Shared-Pointer in C++, aber kein Weak-Pointer)
Built-in-Strings)
vereinfachte Ausgabe,
aktive und passive Klassen, Vererbung,
Module, Precursor-Module

Position

- **Teil A**
  Aspekte
  dynamis
- **Teil B**
  Die Mod
- **Teil C**
  Die ausf
  SLX
- **Teil D**
  Modellie

- **C.1**
  Einführung und Ba
- **C.2**
  Stochastische Pro
- **C.3**
  Vertiefung der SL
- **C.4**
  GPSS-Elemente
- **C.5**
  DISCO-Elemente
- **C.6**
  Basissprache (Erg

1. SLX-Überblick
2. Syntaxkonventionen
3. Elementare Datentypen
4. Klassen
5. Objekte
6. Typ
7. Set
8. Prozeduren
9. Einfache Ausgabe
10. Modellierungselemente in SLX (allgemein)
11. Prozesse und Pucks
12. SLX- Laufzeitsystem (Simulator-Kern)
13. Zeitkonzepte
14. Behandlung von Zeit- und Zustandsereignissen

SLX-Syntaxkonventionen

- blockorientierte Sprache { ... }

- Kommentare: /* ... */
  oder
  Zeilenende-Kommentar //...

- Namensbildung:
  (übliche Konvention)
  - case-sensitiv,
  - Unterstrich erlaubt,
  - qualifizierte Namen (um Eindeutigkeit herzustellen)

- Schlüsselworte

```
public module Module1 {
    int     myCounter1,
            myCounter1A;
}

public module Module2 {
    int     myCounter1,
            myCounter1B;
}

public module Module3 {
    int     myCounter2;

    procedure myProcedure() {
        int     count, myCounter2;
        ...
        count = myCounter1A;          // unique to Module1
        count = myCounter1B;          // unique to Module2
        count = myCounter2;           // local to the current procedure
        count = Module3::myCounter2;  // myCounter2 from Module3
        count = Module1::myCounter1;  // myCounter1 from Module1
        count = Module2::myCounter1;  // myCounter1 from Module2
    }
}
```

Position

- **Teil A**
  Aspekte
  dynamis
- **Teil B**
  Die Moc
- **Teil C**
  Die ausf
  SLX
- **Teil D**
  Modellie

- **C.1**
  Einführung und Ba
- **C.2**
  Stochastische Pro
- **C.3**
  Vertiefung der SL
- **C.4**
  GPSS-Elemente
- **C.5**
  DISCO-Elemente
- **C.6**
  Basissprache (Erg

1. SLX-Überblick
2. Syntaxkonventionen
3. Elementare Datentypen
4. Klassen
5. Objekte
6. Set
7. Prozeduren
8. Einfache Ausgabe
9. Modellierungselemente in SLX (allgemein)
10. Prozesse und Pucks
11. SLX- Laufzeitsystem (Simulator-Kern)
12. Zeitkonzepte
13. Behandlung von Zeit- und Zustandsereignissen

Elementare Datentypen, Anfangswerte

- int (32-bit)
- float = double (64-bit)
- boolean (TRUE oder FALSE)

- alle Variablen besitzen einen Anfangswert
  int : 0,
  float/double : 0.0,
  boolean : FALSE,
  pointer (X) ptr : NULL,
  ...
- auch strukturierte Variablen

- ACHTUNG: keine Zeiger auf elementare Datenstrukturen erlaubt
- Operatoren (aus C übernommen)

```
+     -     *     /     %
<     <=    >     >=
==    !=    =
+=    -=    ++    --
!     NOT   &&    ||
<<    >>    ~     &     ^     |
pow (basis, exponent)
```

- min. bzw. max. Darstellung einer Gleitkomma- bzw. Integerzahl

```
float a, b;
int i, j;
a= INFINITY;
b= -INFINITY;
i= INT_MIN; // -2.147.483.648
j= INT_MAX; // 2.147.483.647
```

SLX-Datentypen (Übersicht)

- Werte, Variablen, Konstante

```
1234
-99
0x10AB                  // hexadecimal
0b100100111             // binary
```

```
int             i, j, k;            // normale Variablen
control int     myCounter;          // Control-Variable
int             CountDown = 10;     // mit nutzerdefiniertem Initialwert
constant int    NCASES = 100;       // Konstante (Wertänderung unmöglich
```

SLX-Datentypen (Übersicht), Variableneigenschaften

- Vordefinierte elementare Datentypen
- Felder
- Aufzählungstypen
- Zeichenketten
- Klassen
- Objektzeiger
- Kollektortyp-Schablone
- Vordefinierte Klassen (Modellbausteine)

- Variablendeklarationen können Präfix besitzen
  public double x = 100.0;

Prefix	Meaning
public	The variable is visible outside its containing scope.
private	The variable is invisible outside its containing scope.
read_only	If visible, the variable can be read, but not written outside its containing scope.
write_only	If visible, the variable can be written, but not read outside its module.
read_write	If visible, the variable can be both read and written outside the its containing scope.
constant	The variable is assigned a mandatory initial value and cannot be changed thereafter.
control	The variable is a state variable. State variables are used in conjunction with SLX's wait until statement, ... At each point in a program where the value of a control variable is changed, SLX inserts a check to see whether any ongoing activity is presently waiting for the variable to attain a given value.
static	For variables used inside procedures or SLX classes, the "static" prefix indicates that a single copy of the variable is shared among all instances of the class or procedure. For non-static variables in classes or procedures, a separate instance of the variable is allocated for each instance of the class or procedure. Variables defined at the module level (outside all classes and procedures) are inherently static.
OEM	Variables with an OEM prefix are hidden from all SLX debugger displays. SLX uses OEM variables in many of its predefined classes to prevent users from snooping at details about which they needn't know.

```
//*****************************************************************************
// EX0039
//*****************************************************************************
public module showPrefix  { // All definitions are public for other modules
   class Example {
       public       int publi ;
       private      int privat ;
       read_only  int readonl;
       write_only int writeonl;
       read_write int readwrit;
  } // class example
} // showPrefix

module executable { // All definitions are private for other modules
    procedure main {

                Example my_example ; // Object of class example
          // Possible
                my_example. publi= 20; // public attribute
                my_example. writeonl = 50; // write_only attribute
                my_example. readwrit = 60 ; // read_write attribute
               print (my_example. publi) "publi : _ \n";
               print (my_example. readwrit ) "readwrit : _ \n";
          // No Chance, because private
                my_example. privat = 30 ;
               •• Semantic error: "privat" is private to "example"
          // Not Possible for read_only
                my_example. readonl = 40;
               •• Semantic error: "my_exa.readonl" is a read_only variable; it cannot be modified
          // Possible for read_only
               print (my_example. readonl ) " readonl : _ \n";
          // Not Possible for write_only
               print (my_example. writeonl ) "writeonl : _ \n";
               •+ {
               •+ slx_start_output(stdout,0,
               •+ "writeonl : _ \n");
               •+ slx_write(my_exa.writeonl);
               •• Semantic error: "my_exa.writeonl" is a write_only variable; you cannot fetch its value
               •+ slx_end_output();
               •+ }
          // Possible for write_only
                my_example. writeonl = 40;
    } //main
} //module executable
```

SLX-Felder

- n-dimensional
- Definition:
- <Basistyp> <Feldname>[d1][d2]...[dn]
- Basistyp muss elementarer Datentyp oder Objektzeiger sein
- Indizierung von 1 bis n
  (nicht wie in C von 0 bis n-1)
- Dimension muss in Def. angegeben werden
  (Variablen sind zulässig)

```
int d = 2;
int f[d];
f[2] = 3;
```

- Dimension darf nur bei Verwendung als Prozedur-Parameter entfallen
-

```
int g[2][2] = { {1,2}, {3} };
procedure  p(int arg[*]) {}
```

- Angabe einer **initialen** Belegung für **jede** Dimension möglich
  (sonst Vorbelegung mit Standardwert)

SLX-Aufzählungstypen

```c
// C
#include <stdio.h>
enum Color {
    red, green, blue
};
int main() {
    Color c = green;
    int i = c;
    printf("%d",i);
}
```

```
// SLX
type Color enum {
    red, green, blue
};
```

```
procedure main() {
    Color c = blue;
    c = first Color; // red
    c = predecessor(c); // NONE
    int i = c;
•• Semantic error: an int is required
here; "c" is a Color
}
```

- Definition in SLX:

```
type <Name> enum {
  <Wert1>, <Werte2>, ...
}
```

- Standardwert:
  NONE-Literal
- Werte eines Enum-Typs per
  for-Konstrukt iterierbar
- keine Verwendung von enum-Wert
  als int-Wert erlaubt

SLX-Zeichenketten

```
procedure p(string(*) arg) {}
```

```
string(5) h = "hello";
```

```
string(3) h2 = "hello"; // hel
```

```
string(5) h3 = h2 cat "lo"; // hello
```

```
int i = length(h3);
```

```
string(1) a = ascii(65); // a = "A"
```

```
procedure p(string(*) arg) {}
```

- Definition:
  string(<Länge>) <Name>
- Initial-Wert:
  "" (leere Zeichenkette)
- vordefinierte Funktionen:
  cat, substring, length,
  ascii *(ascii-nach-string-Konv.)*,
  str_ivalue (*string-nach-int-Konv.*)
- keine dynamischen Größenänderungen
  Länge muss angegeben werden
  (nur Integer-Konstanten erlaubt)
- Länge darf nur bei Verwendung als Prozedur-Parameter entfallen

weitere Operationen ...

Position

- **Teil A**
  Aspekte
  dynamis
- **Teil B**
  Die Mod
- **Teil C**
  Die ausf
  SLX
- **Teil D**
  Modellie

- **C.1**
  Einführung und Ba
- **C.2**
  Stochastische Pro
- **C.3**
  Vertiefung der SL
- **C.4**
  GPSS-Elemente
- **C.5**
  DISCO-Elemente
- **C.6**
  Basissprache (Erg

1. SLX-Überblick
2. Syntaxkonventionen
3. Elementare Datentypen
4. Klassen
5. Objekte
6. Typ
7. Set
8. Prozeduren
9. Einfache Ausgabe
10. Modellierungselemente in SLX (allgemein)
11. Prozesse und Pucks
12. SLX- Laufzeitsystem (Simulator-Kern)
13. Zeitkonzepte
14. Behandlung von Zeit- und Zustandsereignissen

SLX-Klassen

- Beispiel:

```
class X (int i) {

        initial {no=i;}

        int no;

        procedure p() {...}

        actions {...}

}
```

Definition:

```
[<Prefix>] class <Name>
        [(<Constructor_Parameter>)] {

        <Attribute>;
        ...
        <Procedure>;
        ...
        <Property>;
        ...
}
```

Klasse gegeben durch optionale Bestandteile

- Attribute (Klassenparameter)
- Prozeduren
- Properties:
  actions, initial, final, clear, report

SLX unterscheidet
aktive und passive Klassen
Präfix= passive
(Default= active)

Aktive und Passive Klassen

Verdeckte Superklasse

- jedes Objekt (einer beliebigen Klasse) verfügt über das Attribut
  use_count
  wird vom SLX-Laufzeitsystem verwendet

- Wertermittlung use_count() liefert **integer**

- Bei jedem Kopieren der Adresse eines Objektes auf eine **Pointer**-Variable wird use_count inkrementiert.

- wird der Pointer-Variablen ein neuer Wert zugewiesen (zeigt also nicht mehr auf das Objekt), so wird use count wieder dekrementiert.

- Die Vernichtung des Objektes wird solange ausgesetzt, bis der use_count des Objektes auf den Wert **Null** dekrementiert wurde.

wichtig für Umgang mit Pointern:
Zeiger können niemals „ins Blaue“ zeigen (Programmsicherheit)

Klassenparameter und Property intial

```
class X (int j) {
   int i;

   initial {
         i = j;
   }
}

class Y (int j) {
   int i = j;
}

class Z (int i) {
   int i;
   •• Semantic error: "i" has been
   previously defined in the current
   scope
}
```

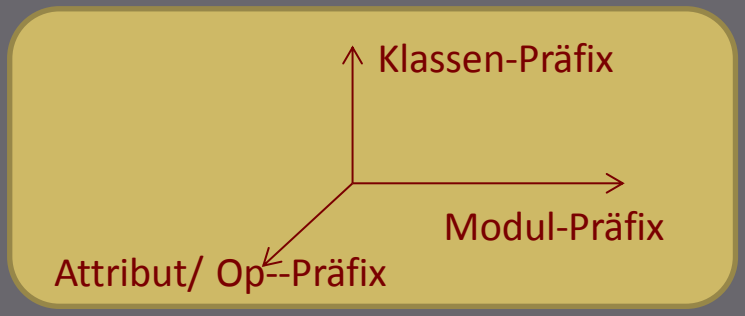
- Parameterangaben hinter dem Klassennamen
- Impliziter Konstruktor ruft initial-Property zur initialen nutzerdefinierten Belegung der Objekt-Attribute
- höchstens ein initial

SLX kennt statische und dynamische Instanzen/Objekte

Sichtbarkeit von Klassenattributen und Operationen (2)

```
class X {
                    int i; // public
            private int j;

                    procedure p() { }
            private procedure q() { }
}
```

- Sichtbarkeit von Attributen und Prozeduren:
  - public (Standardfall)
  - protected
  - private




```
Module M1 {

}
```

Standardfall private

anders als in anderen Sprachen

- z.B. C++
  - class: private
  - struct/union: public

```
private class X {
                    int i; // private
            public  int j;
}
```

Position

- **Teil A**
  Aspekte
  dynamis
- **Teil B**
  Die Mod
- **Teil C**
  Die ausf
  SLX
- **Teil D**
  Modellie

- **C.1**
  Einführung und Ba
- **C.2**
  Stochastische Pro
- **C.3**
  Vertiefung der SL
- **C.4**
  GPSS-Elemente
- **C.5**
  DISCO-Elemente
- **C.6**
  Basissprache (Erg

1. SLX-Überblick
2. Syntaxkonventionen
3. Elementare Datentypen
4. Klassen
5. Objekte
6. Typ
7. Set
8. Prozeduren
9. Einfache Ausgabe
10. Modellierungselemente in SLX (allgemein)
11. Prozesse und Pucks
12. SLX- Laufzeitsystem (Simulator-Kern)
13. Zeitkonzepte
14. Behandlung von Zeit- und Zustandsereignissen

Objekte

- existieren (wie in C) entweder

  (1) lokal auf dem Stack
  mit automatischer Vernichtung bei
  Verlassen des Gültigkeitsbereiches

  oder

  (2) global auf dem Heap

```
class X (int j) {
    int i;
    Y y;

    initial {
            i = j;
    }
}

class Y { }

X global_x(1);

procedure main() {
            X local_x(1);
}
```

Aufgabe des impliziten Konstruktors

Speicher für Objekt entspr. des Klassenlayouts bereitstellen → Attributinitialisierung, Init-Property

Objektzeiger

- Zeiger-Definition für bestimmte Objekte:
  pointer (<Klasse>) <Name>
- Zeiger-Definition für bel. Objekte:
  pointer (*) <Name>
- Typ des Zeigers muss eine Klasse sein
  (insbes. kein Zeiger auf Zeiger
  und kein Zeiger auf Werte von
  Datentypen)
- Zeiger-Wert: entspr. Objektadresse
  Standardwert: **NULL**-Literal

```
class X (int j) {
      int i;

      initial {
            i = j;
      }
}

X global_x(1);

procedure main() {
     pointer(X) px = &global_x;

     px = new X(2);
}
```

Objektzeiger

Zeiger-Operatoren (übernommen aus C):

Gegeben sei die Klassendefinitionen ClassX {}, ClassY{} und
ein Objekt x von ClassX,
definiert per ClassX() x sowie
ein Zeiger y auf ein ClassY-Objekt,
mit **pointer**(ClassY) y = **new** ClassY();

- Zeiger auf das Objekt x
  &x

```
pointer (ClassX) px= &x;
```

- Objekt am Zeiger y
  *y

```
procedure  proc ( classY c) {…}; //Def
```

```
proc ( *y);                          //Aufruf
```

- Wert des Attributes i, wobei x ein Objekt ist
  x.i

- Wert des Attributes i, wobei y ein Zeiger ist
  y->i

Objektzeiger

```
class ClassX (int j) {
        int i;
        initial {
                i = j;
    }
}

class ClassY {
    int m;
}

procedure main() {
    pointer(*) u = new ClassX(1);
    int k = u->m;
    •• Execution error at time 0:
    "u" points to an object of class X,
    which has no "m"
}
```

- Universalzeiger

  pointer(*) zeiger;

- Zugriff auf nicht vorhandene Attribute über Universal-Zeiger?
  - Compilierfehler falls keine Klasse das Attribut definiert
  - Laufzeitfehler sonst

Referenzzählung

```
pointer(X) x1 = new X(1);
pointer(X) x2;
// RZ vom X-Objekt = 1

x2 = x1;
// RZ vom X-Objekt = 2

x1 = NULL;
// RZ vom X-Objekt = 1

x2 = NULL;
// RZ vom X-Objekt = 0
// X wird jetzt vernichtet
```

- jedes Objekt hat einen **Referenzzähler** (RZ),

  in SLX use_count

- **Wertaktualisierung** des RZ ist an Zuweisungen für Pointer-Variablen durch den **Assignment-Operator** (=) gebunden

- Schritte:
  1. Zähler des aktuell referenzierten Objektes wird verringert
  2. Zähler des neu referenzierten Objektes wird erhöht

- entspricht shared_ptr<T> aus BOOST/C++

Destruktor: Property final

```
class Y {
   int i;
   final {
      print "Y destructor\n";
   }
}
class X (int j) {
   int i = j;
   pointer(Y) y;

   initial {
      y = new Y();
   }
   final {
     print(i) "X(_) destructor\n";
   }
}
```

- automatische Speicherfreigabe
  über Referenzzählung
- manuell: **destroy** (zeiger)
  (Wirkung, nur wenn zeiger alleiniger Zeiger ist)

Anwendung von **final** zur Protokollierung

```
pointer(X) x1 = new X(1);
pointer(X) x2 = new X(2);
      print "before x1 = x2\n";
x1 = x2;
      print "after x1 = x2\n";
destroy x1; // ???
     print "after destroy x1\n";
destroy x2;
```

AUSGABE:

```
Execution begins
        before x1 = x2
        X( 1      ) destructor
        Y destructor
        after x1 = x2
        after destroy x1
        X( 2      ) destructor
        Y destructor
Execution complete
```

?

TEST-Aufgabe

Position

- **Teil A**
  Aspekte
  dynamis
- **Teil B**
  Die Mod
- **Teil C**
  Die aust
  SLX
- **Teil D**
  Modellie

- **C.1**
  Einführung und Ba
- **C.2**
  Stochastische Pro
- **C.3**
  Vertiefung der SL
- **C.4**
  GPSS-Elemente
- **C.5**
  DISCO-Elemente
- **C.6**
  Basissprache (Erg

1. SLX-Überblick
2. Syntaxkonventionen
3. Elementare Datentypen
4. Klassen
5. Objekte
6. Typ
7. Set
8. Prozeduren
9. Einfache Ausgabe
10. Modellierungselemente in SLX (allgemein)
11. Prozesse und Pucks
12. SLX- Laufzeitsystem (Simulator-Kern)
13. Zeitkonzepte
14. Behandlung von Zeit- und Zustandsereignissen

Dynamische Typbestimmung

```
pointer(*) u = new X(1);

if (type(*u) == type X) {
    // object, referenced by u, is of type X
                ...
}
if (type(*fahrzeug) == type Fahrrad) {
    print(fahrzeug->rad_anzahl)
                "Fahrrad mit _ Rädern\n";
}
```

- a) Typ eines Ausdrucks:

  type (<Ausdruck>)

- b) Typ einer Klasse:

  type <Klasse>

  Ergebnis:

  Typ-Deskriptor

```
if (type (TRUE) == type boolean)
            print "bool";

if (type (1+3) == type int)
            print "int";
```

ACHTUNG: Unterschied

type(*fahrzeug)

type (fahrzeug)

Position

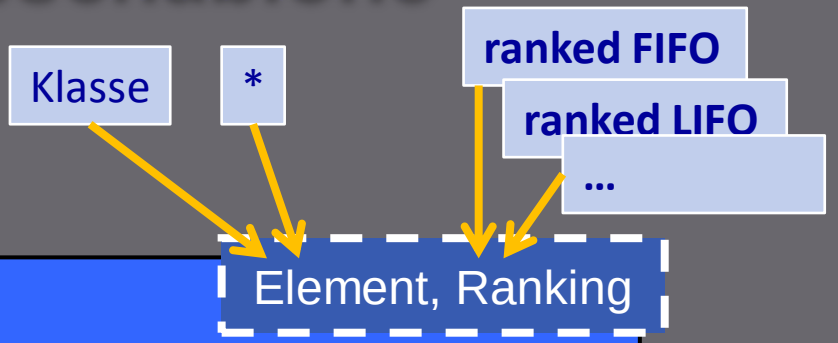
- **Teil A**
  Aspekte
  dynamis
- **Teil B**
  Die Mod
- **Teil C**
  Die aust
  SLX
- **Teil D**
  Modellie

- **C.1**
  Einführung und Ba
- **C.2**
  Stochastische Pro
- **C.3**
  Vertiefung der SL
- **C.4**
  GPSS-Elemente
- **C.5**
  DISCO-Elemente
- **C.6**
  Basissprache (Erg

1. SLX-Überblick
2. Syntaxkonventionen
3. Elementare Datentypen
4. Klassen
5. Objekte
6. Typ
7. Set
8. Prozeduren
9. Einfache Ausgabe
10. Modellierungselemente in SLX (allgemein)
11. Prozesse und Pucks
12. SLX- Laufzeitsystem (Simulator-Kern)
13. Zeitkonzepte
14. Behandlung von Zeit- und Zustandsereignissen

Set als SLX-Typschablone

Zur Definition von Objekt-Kollektionen beliebiger Klassen

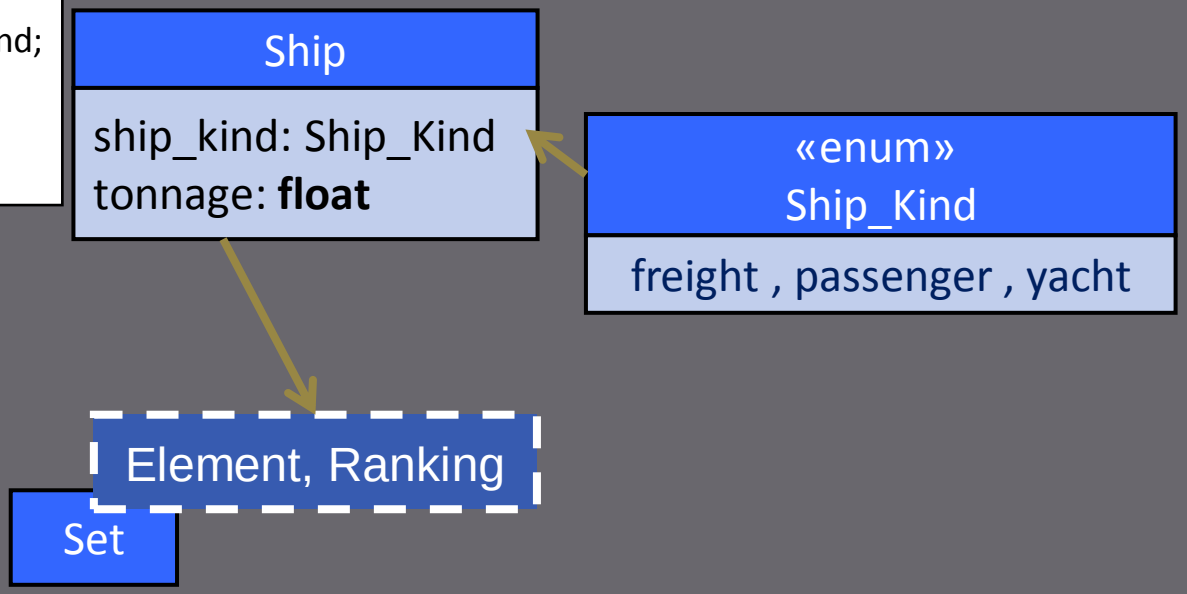


Set	
control int size	Erklärung später
place (...)	*einfügen einer Element-Instanz in Set evtl. an eine bestimmte Position*
remove (...)	*entnehmen einer Element-Instanz*
size (...): **int**	*aktuelle Elementanzahl von Set*
position (...): **pointer** (Element)	*Elementinstanz auf einer bestimmten Position von Set*
for_Iterator (...)	*Anwendung einer Folge von Anweisungen für jedes oder ausgewählte Elemente von Set*
first (...)	*erstes Element*
last (...)	*letztes Element*
succuessor (...): **pointer** (Element)	*Nachfolger-Element*
Predecessor (...): **pointer** (Element)	*Vorgänger-Element*
contains (...): **boolean**	*Test, ob Set ein bestimmtes Element enthält*
is_in (...): **boolean**	*Test, ob sich ein Element in einer bestimmten Menge befindet*
is_not_in (...): **boolean**	*Test, ob sich ein Element in einer bestimmten Menge befindet*
report	Erklärung später
clear	
final	

Achtung: Operationen haben komplexe Signaturen und werden so nicht angeboten. Stattdessen gibt es flexible Schlüsselwort-gesteuerte Anweisungen

Beispiel

```
class Ship {
    enum { freight , passenger , yacht } ship_kind;
    float tonnage;
}
```




```
set ( Ship ) harbor1;                    // homogene Kollektion nach FIFO

set ( Ship ) ranked LIFO harbor2;        // homogene Kollektion nach LIFO

set ( Ship ) ranked ( ascending ship_kind, descending tonnage)
             harbor3;                    // aufsteigend nach Typ,
                                         // absteigend nach Tonnage

set (*) collector;                       //unranked heterogenous set
```

vier verschiedene Set-Objekte

~ d.h.
vier zunächst leere Kollektionen

Mengentypen: set

```
class X {
            int a;
            int b;
}

set(X) xs;

set(X) ranked LIFO xsl;

set(X) ranked
        (ascending a, descending b) s;
```

Inhalt einer Kollektion sind immer nur Zeiger auf Element (hier: Ship-Zeiger oder X-Zeiger)

Damit kann eine Element-Instanz in mehr als einem Set-Objekt gleichzeitig enthalten sein

- Definition (homogenes Set): set(<Klasse>) <Name>
- Definition (universelles Set): set(*) <Name>
- Sortierung einstellbar:
  - FIFO (Standard),
  - LIFO,
  - aufsteigend/absteigend nach Attributen (nur für homogene Sets)

Set-Operationen als Anweisungen (1)

```
set(X) xs;
set(X) ranked LIFO xsl;

pointer(X) x1 = new X();

place x1 into xs;
```

```
int i = position(x1) in xs;
remove x1 from xs;

empty xs;
empty set xs;
```

äquivalent

```
pointer(X) x;
for (x = each X in xs) { }
for (x = each object in xs) { }
```

- Einfügen
  place <Pointer> **into** <Set>
- Einfügen an Position
  place <Pointer> **into** <Set> **after** <Pointer>
- Position ermitteln
  int <Position> = **position**(<Pointer>) **in** <Set>
- Entfernen
  remove <Pointer> **from** <Set>
- Leeren
  empty <Set>
- Iterierte Bearbeitung von
  - ausschließlich Objekte einer bestimmten Klasse/Ableitung:
    for (<Pointer> = **each** <Klasse> **in** <Set>) {...}
  - allen Objekten vom Set-Objekt:
    for (<Pointer> = **each object in** <Set>) {...}