
# 5. Symbolliste




Hanspeter Mössenböck, Institut für Systemsoftware, JKU

# *Aufgaben der Symbolliste*


**1. Speicherung aller deklarierten Namen und ihrer Eigenschaften**

- Typ
- Wert (bei Konstanten)
- Adresse (bei Variablen, Feldern und Methoden)
- Parameter (bei Methoden)

**2. Suchen eines Namens und seiner Attribute**

- Abbildung: Name $\Rightarrow$ (Typ, Wert, Adresse, ...)

**3. Verwaltung von Typen**

- einfache Typen (*int*, *char*)
- strukturierte Typen (Arrays, Klassen)

**4. Verwaltung von Gültigkeitsbereichen (Scopes)**

**Inhalt der Symbolliste**

- Objektknoten: Informationen über deklarierte Namen
- Strukturknoten: Informationen über Typstrukturen
- Scopeknoten: Verwaltung der Gültigkeitsbereiche von Namen

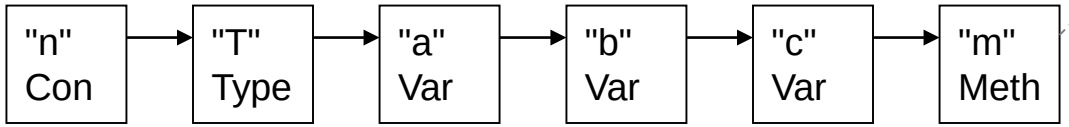
=> Es bietet sich eine dynamische Datenstruktur an (Lineare Liste, Binärbaum, Hashtabelle)

# *Symbolliste als lineare Liste*


**Gegeben: folgende Deklarationen**

```
final int n = 10;
class T { ... }
int a, b, c;
void m() { ... }
```

**Symbolliste als lineare Liste**




für jeden deklarierten Namen gibt es einen Objektknoten

\+ einfach

\+ enthält gleichzeitig Deklarationsreihenfolge (wichtig falls Adressvergabe erst später erfolgt)

\- lange Suchzeiten wenn es viele Deklarationen gibt

**Einfachste Schnittstelle**

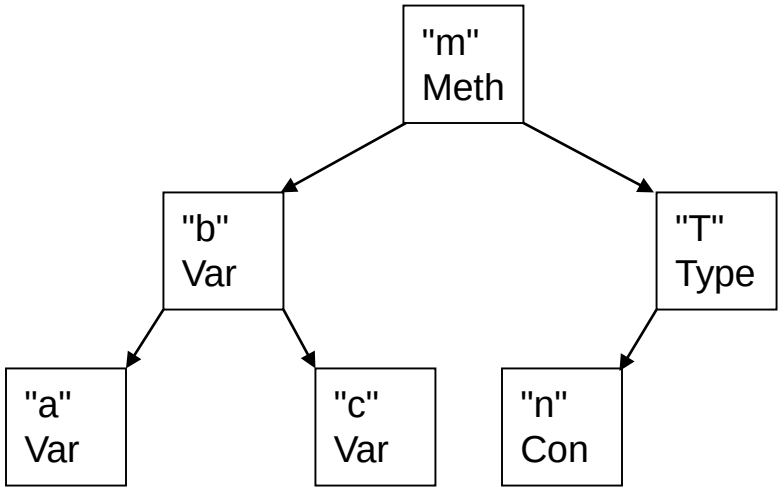
```
public class Tab {
   public static Obj insert (String name, ...);
   public static Obj find (String name);
}
```

# *Symbolliste als Binärbaum*

## Deklarationen

```
final int n = 10;
class T { ... }
int a, b, c;
void m() { ... }
```

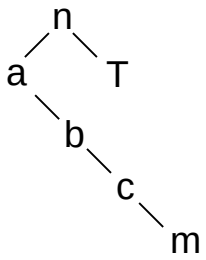
## Symbolliste als binärer Suchbaum




\+ schnellere Suchzeiten

\- kann degenerieren, wenn nicht balanciert
\- mehr Speicherbedarf
\- Deklarationsreihenfolge geht verloren

Nur bei sehr vielen Deklarationen sinnvoll

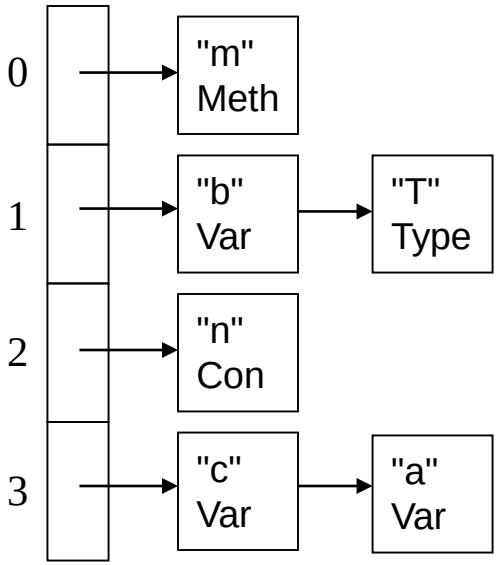


# *Symbolliste als Hashtabelle*

## Deklarationen

```
final int n = 10;
class T { ... }
int a, b, c;
void m() { ... }
```

## Symbolliste als Hashtabelle




\+ schnelle Suchzeiten

\- komplizierter als lineare Liste
\- Deklarationsreihenfolge geht verloren

Für unserer Zwecke reicht eine lineare Liste

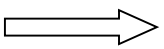
- Jeder Scope braucht ohnehin eine eigene Datenstr.
- Ein Scope hat kaum mehr als 10 Namen


# 5. Symbolliste

# *Objektknoten*

Jeder deklarierte Name wird in einem Objektknoten gespeichert

```
int a;
```



*Obj*

"a"
...

**Arten von Objekten in MicroJava**

- Konstanten
- Variablen und Felder
- Typen
- Methoden

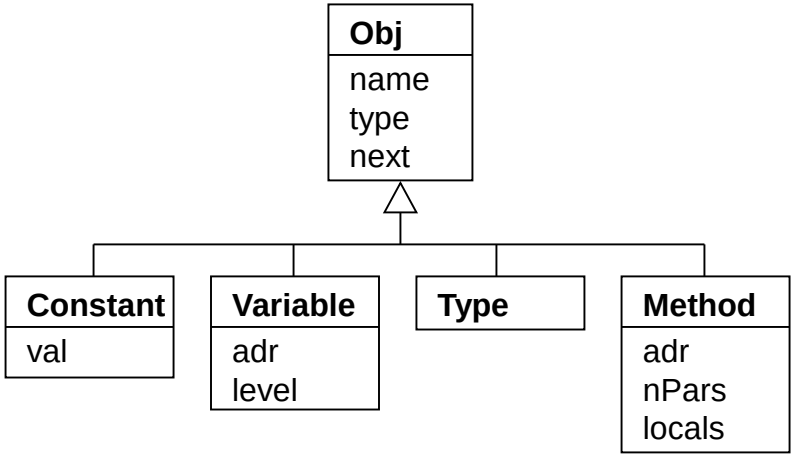
```
static final int
   Con   = 0,
   Var   = 1,
   Type  = 2,
   Meth  = 3;
```

**Was muss über Objekte gespeichert werden?**

- für alle Objekte — Name, Objektart, Typ, Next-Zeiger
- für Konstanten — Wert
- für Variablen — Adresse, Deklarationsstufe
- für Typen — -
- für Methoden — Adresse, Parameterzahl, Liste formaler Parameter

# *Mögliche objektorientierte Struktur*

**Folgende Klassenhierarchie wäre naheliegend**




Ist aber umständlich wegen ständiger Type Casts

```
Obj obj = Tab.find("x");
if (obj instanceof Variable) {
    Variable v = (Variable)obj;
    v.adr = ...;
    v.level = ...;
}
```

Daher "flache Implementierung": Alle Felder stehen in derselben Klasse.
Ist vertretbar, da

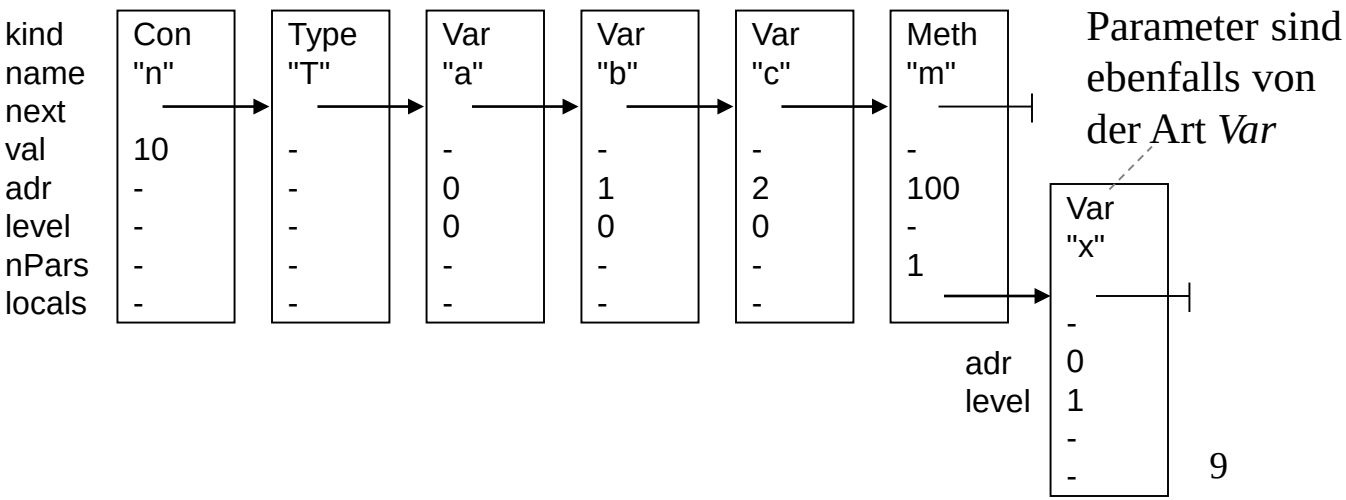
- Anzahl der Objekt-Varianten fix (keine Erweiterbarkeit erforderlich)
- Keine dynamische Bindung nötig

# *Klasse Obj*

```
class Obj {
   static final int Con = 0, Var = 1, Type = 2, Meth = 3;
   int      kind;       // Con, Var, Type, Meth
   String   name;
   Struct   type;
   Obj      next;
   int      val;        // Con: value
   int      adr;        // Var, Meth: address
   int      level;      // Var: 0 = global, 1 = local
   int      nPars;      // Meth: number of parameters
   Obj      locals;     // Meth: parameters and local objects
}
```

**Beispiel**

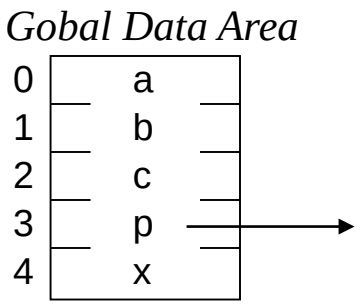
```
final int n = 10;
class T { ... }
int a, b, c;
void m(int x) { ... }
```



# *Globale Variablen*

**Werden in der *Global Data Area* der MicroJava VM gespeichert**

```
program Prog
   int a, b;
   char c;
   Person p;
   int x;
{ ... }
```

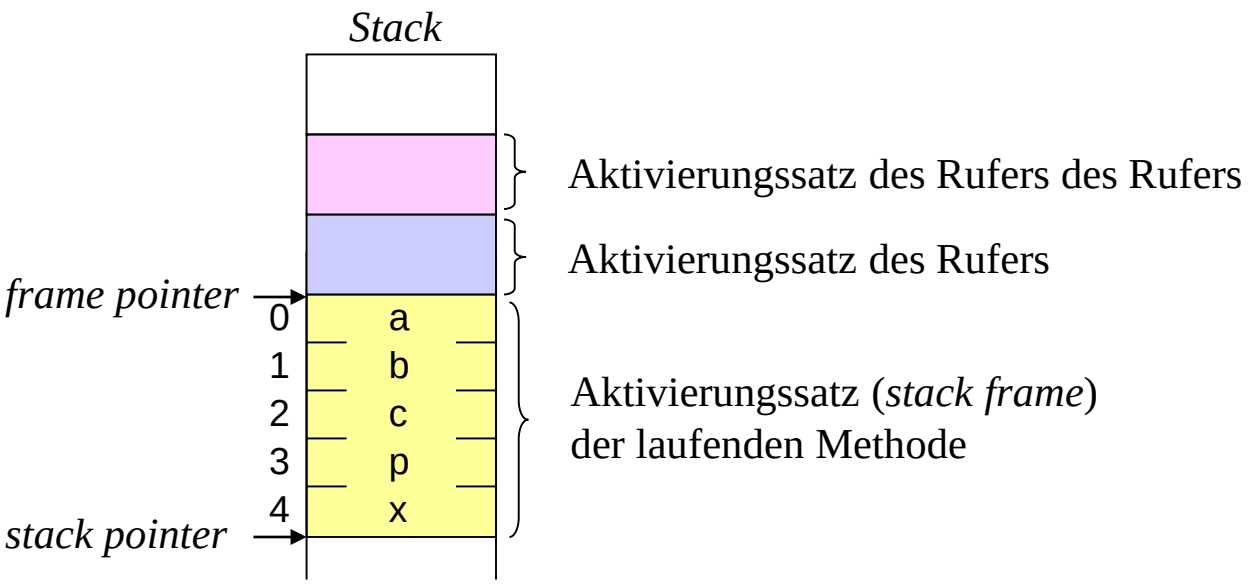



- Jede Variable belegt in MicroJava 1 Wort (4 Bytes)
- Adressen sind Wortnummern relativ zur Global Data Area
- Adressen werden fortlaufend in Deklarationsreihenfolge vergeben

# *Lokale Variablen*

**Werden in einem "Aktivierungssatz" (Frame) am Methodenkeller gespeichert**

```
void foo()
    int a, b;
    char c;
    Person p;
    int x;
{ ... }
```




- Jede Variable belegt 1 Wort (4 Bytes)
- Adressen sind Wortnummern relativ zum *frame pointer*
- Adressen werden fortlaufend in Deklarationsreihenfolge vergeben

# *Eintragen von Namen in die Symbolliste*


**Bei jeder Deklaration wird folgende Methode aufgerufen**

```
Obj obj = Tab.insert(kind, name, type);
```

- erzeugt neuen Objektknoten mit *kind*, *name*, *type*
- prüft, ob *name* bereits deklariert ist (wenn ja, Fehlermeldung)
- vergibt fortlaufende Adressen für Variablen und Felder
- trägt bei Variablen die Deklarationsstufe ein (0 = global, 1 = lokal)
- hängt den neuen Knoten ans Ende der Symbolliste
- gibt den neuen Knoten als Funktionswert zurück

**Beispiel für den Aufruf von *insert()***

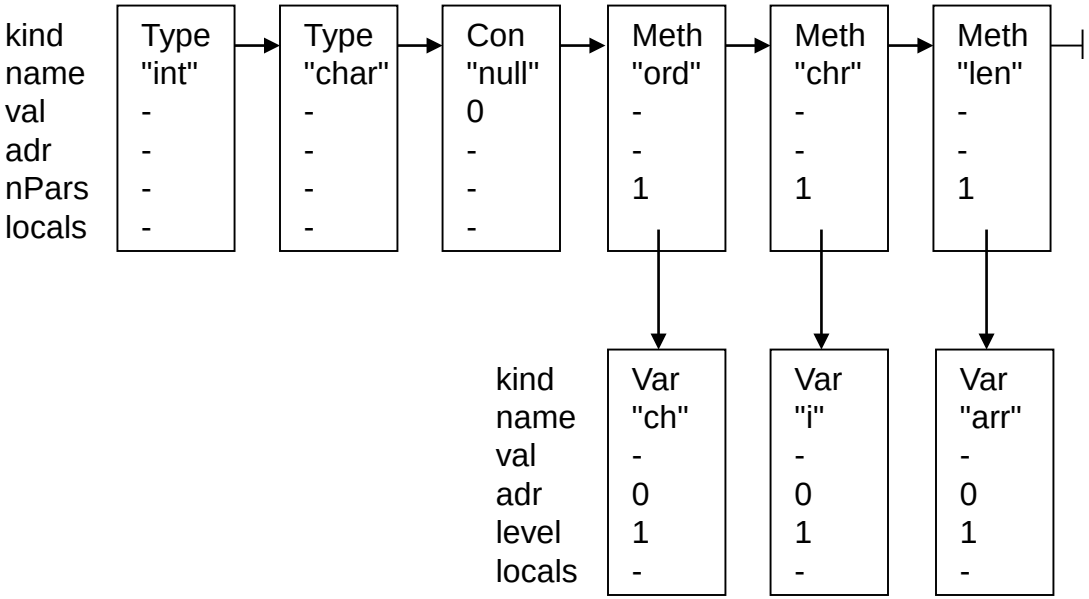
```
VarDecl                    (. Struct type; String name; .)
=  Type<↑type>
   ident<↑name>            (. Tab.insert(Obj.Var, name, type); .)
   { "," ident<↑name>      (. Tab.insert(Obj.Var, name, type); .)
   }
   ";" .
```

# *Vordeklarierte Namen*

**Welche Namen sind in MicroJava vordeklariert?**

- Standardtypen: int, char
- Standardkonstanten: null
- Standardmethoden: ord(ch), chr(i), len(arr)

**Vordeklarierte Namen werden in der Symbolliste gespeichert**



# *Alternative: als Schlüsselwörter*


**_int_ und _char_ könnten auch als Schlüsselwörter implementiert werden**

erfordert aber Sonderbehandlung in der Grammatik

```
Type<↑type>
=   ident<↑name>   (. Obj x = Tab.find(name); type = x.type; .)
|   "int"          (. type = Tab.intType; .)
|   "char"         (. type = Tab.charType; .) .
```

**Es ist einfacher, sie in der Symbolliste vorzudeklarieren**

```
Type<↑type>
=  ident<↑name>    (. Obj x = Tab.find(name); type = x.type; .).
```

\+ einheitliche Behandlung vordefinierter und benutzerdefinierter Typen
\- jemand kann "int" als benutzerdefinierten Typ überschreiben


# 5. Symbolliste

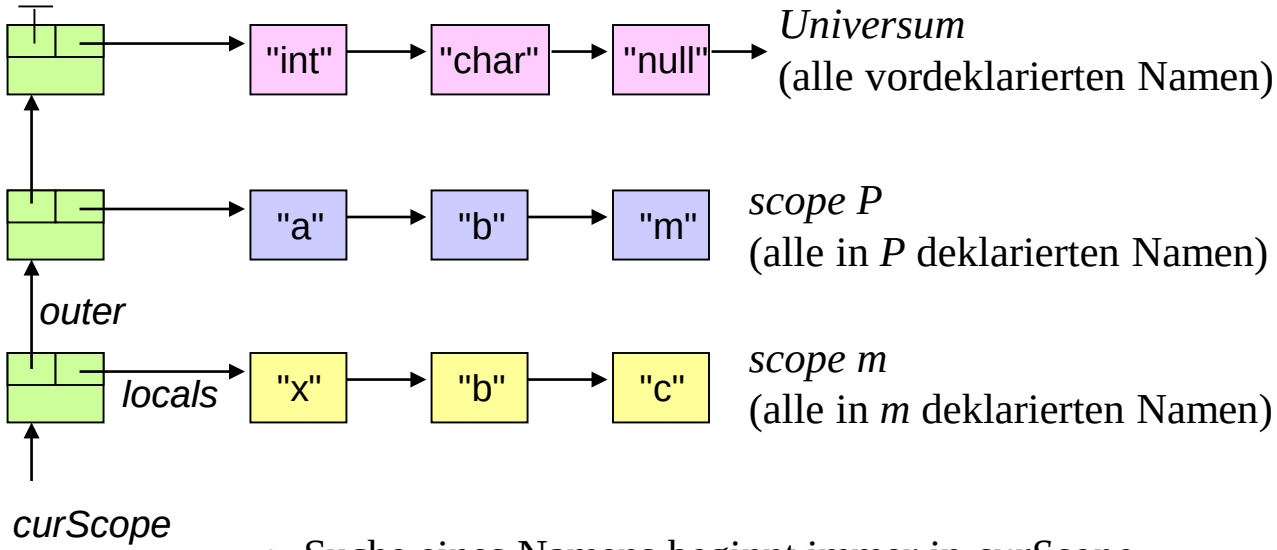
# *Scope = Gültigkeitsbereich für Namen*


**Jeweils 1 Scope für**

- Programm — enthält globale Namen
- jede Methode — enthält lokale Namen
- jede Klasse — enthält Felder
- "Universum" — enthält vordeklarierte Namen

**Beispiel**

```
program P
   int a, b;
{
   void m (int x)
      int b, c;
   {
      ...
   }
   ...
}
```




- Suche eines Namens beginnt immer in *curScope*
- Wenn nicht gefunden, im nächstäußeren Scope suchen
- Beispiel: Suche von *b*, *a* und *int*

# *Scope-Knoten*


```
class Scope {
   Scope  outer;    // zum nächstäußeren Scope
   Obj    locals;   // zur Liste der Objekte in diesem Scope
   int    nVars;    // Anzahl der Variablen in diesem Scope (zur Adressvergabe)
}
```

## Erzeugen von Scopes

```
static void openScope() { // in class Tab
   Scope s = new Scope();
   s.outer = curScope;
   curScope = s;
   curLevel++;
}
```

- aufgerufen am Beginn einer Methode oder Klasse
- verkettet neuen Scope mit den bestehenden
- neuer Scope wird *curScope*
- *Tab.insert()* trägt Namen immer in *curScope* ein
- *curScope*, *curLevel*: globale Variablen in *Tab*

## Schließen von Scopes

```
static void closeScope() { // in class Tab
   curScope = curScope.outer;
   curLevel--;
}
```

- aufgerufen am Ende einer Methode oder Klasse
- macht nächstäußeren Scope zu *curScope*

# *Öffnen und Schließen von Scopes*


```
MethodDecl                (. Struct type; String name; .)
=  Type<↑type>
   ident<↑name>           (.  curMethod = Tab.insert(Obj.Meth, name, type);
                              Tab.openScope(); .)
   "(" ... ")"
   ...
   "{"                    (. curMethod.locals = Tab.curScope.locals; .)
   ...
   "}"                    (.  Tab.closeScope(); .)
   .
```

**Beachte**

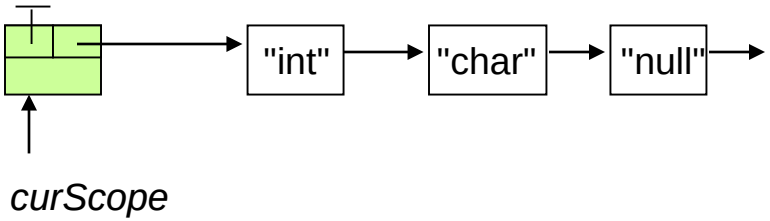
- Methodenname wird noch in äußeren Scope eingetragen
- *curMethod* ist eine globale Variable vom Typ *Obj*
- Nach Abarbeitung der Deklarationen, werden lokale Objekte des Scopes an *curMethod.locals* gehängt
- Auch bei Klassen wird ein Scope geöffnet und wieder geschlossen

# *Einfügen von Namen in Scopes*


Namen werden bei ihrer Deklaration immer in *curScope* eingetragen

```
class Tab {
   static Scope curScope;    // Zeiger auf aktuellen Scope
   static int curLevel;      // aktuelle Deklarationsstufe (0 = global, 1 = lokal)
   ...
   public static Obj insert (int kind, String name, Struct type) {
      //--- erzeugen
      Obj obj = new Obj(kind, name, type);
      if (kind == Obj.Var) {
         obj.adr = curScope.nVars; curScope.nVars++;
         obj.level = curLevel;
      }
      //--- einfügen
      Obj p = curScope.locals, last = null;
      while (p != null) {
         if (p.name.equals(name)) error(name + " declared twice");
         last = p; p = p.next;
      }
      if (last == null) curScope.locals = obj; else last.next = obj;
      return obj;
   }
   ...
}
```

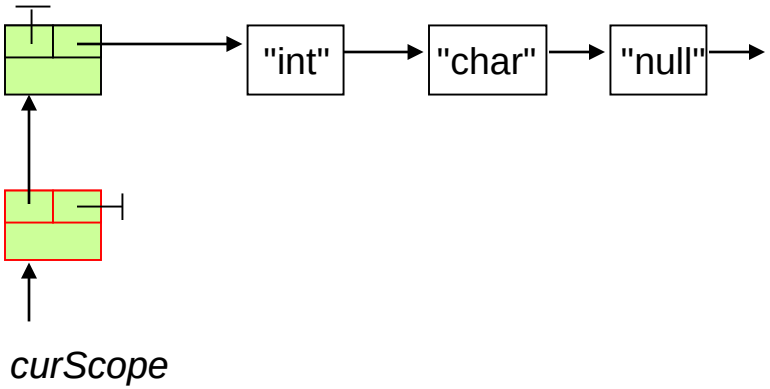
# *Beispiel*



# *Beispiel*

program P

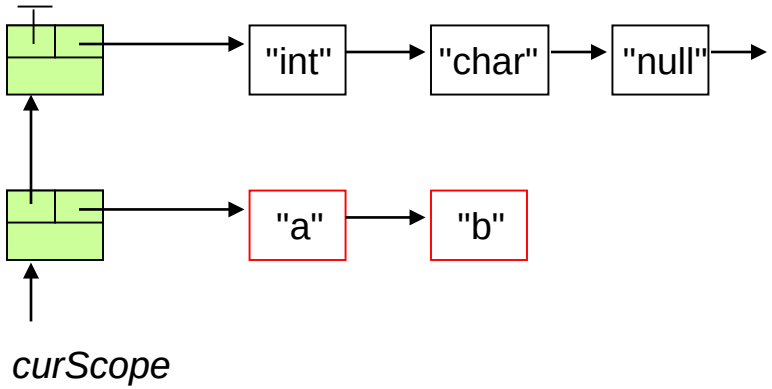
Tab.openScope();



# *Beispiel*

```
program P
    int a, b;
{
```

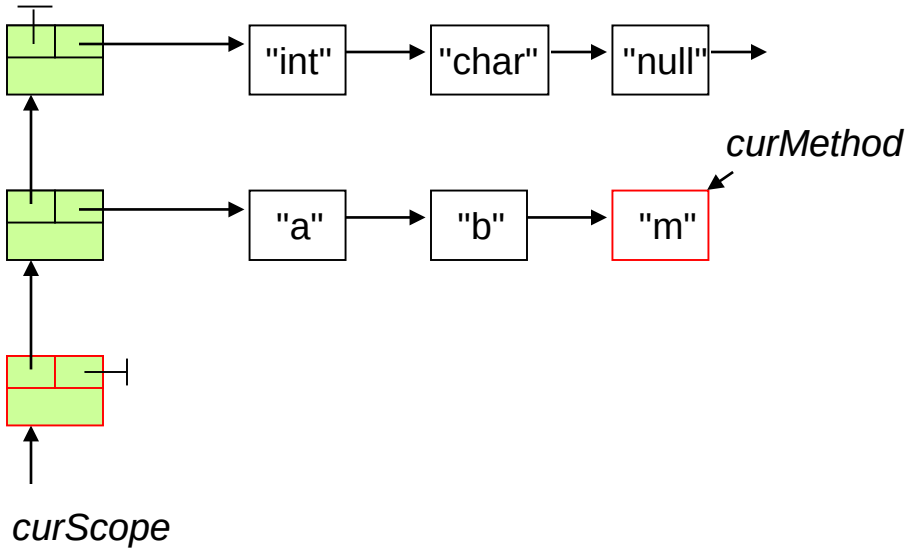
```
Tab.insert(..., "a", ...);
Tab.insert(..., "b", ...);
```



# *Beispiel*

```
program P
    int a, b;
{
    void m()
```

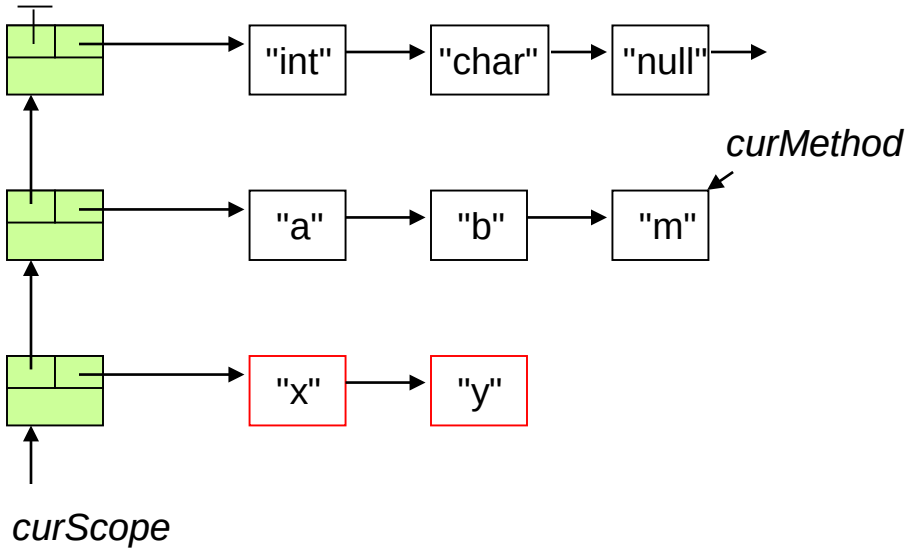
```
Tab.insert(..., "m", ...);
Tab.openScope();
```



# *Beispiel*

```
program P
    int a, b;
{
    void m()
        int x, y;
```

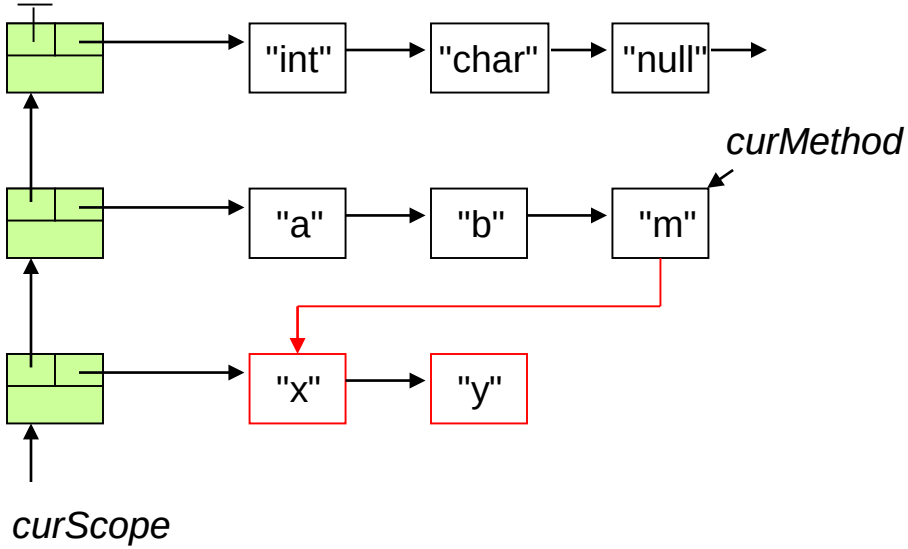
```
Tab.insert(..., "x", ...);
Tab.insert(..., "y", ...);
```



# *Beispiel*

```
program P
    int a, b;
{
    void m()
        int x, y;
    {
```

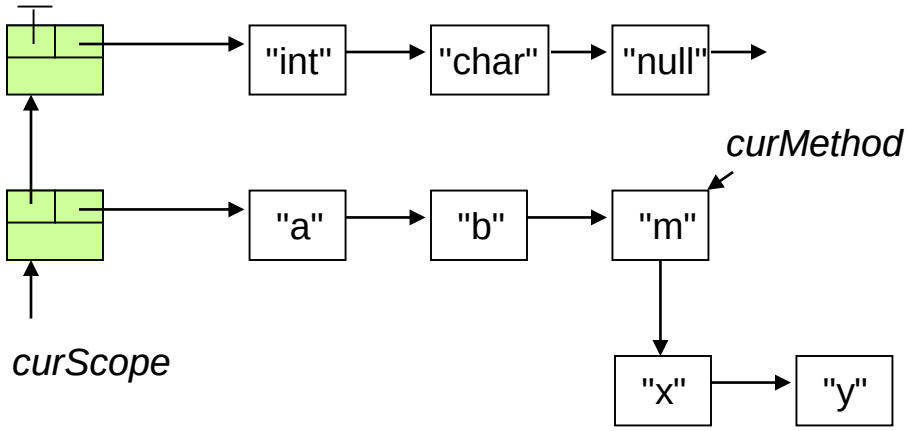
curMethod.locals =
Tab.curScope.locals



# *Beispiel*

```
program P
   int a, b;
{
   void m()
      int x, y;
   {
      ...
   }
```

```
Tab.closeScope();
```

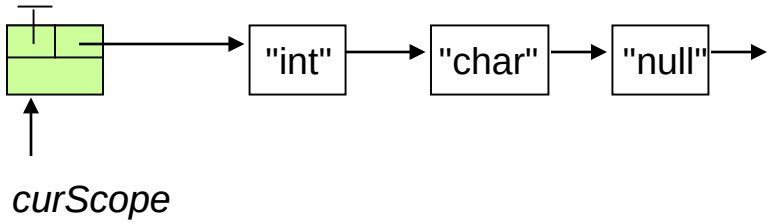



Würde eine weitere Methode folgen,
würde ein neuer Scope dafür angelegt und am Methodenende wieder geschlossen
=> kellerartige Verwaltung von Scopes

# *Beispiel*

```
program P
   int a, b;
{
   void m()
      int x, y;
   {
      ...
   }
   ...
}
```

Tab.closeScope();



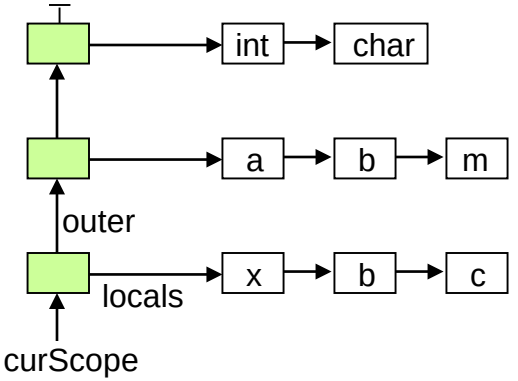
# *Suchen von Namen in der Symbolliste*

**Bei jedem Auftreten eines Namens wird folgende Methode aufgerufen**

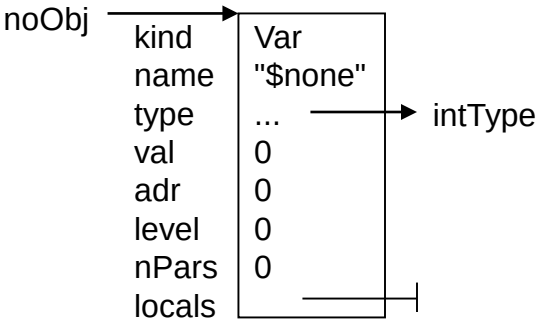
```
Obj obj = Tab.find(name);
```

- Suche beginnt in *curScope*
- Falls nicht gefunden, im nächstäußeren Scope weitersuchen

```
public static Obj find (String name) {
   for (Scope s = curScope; s != null; s = s.outer)
      for (Obj p = s.locals; p != null; p = p.next)
         if (p.name.equals(name)) return p;
   error(name + " is undeclared");
   return noObj;
}
```




**Falls Name nicht gefunden, *noObj* liefern**




- vordefiniertes Dummy-Objekt
- besser als *null*, weil es sonst Folgefehler (Exceptions) gibt


# 5. Symbolliste

# *Typen*


**Objekte haben einen Typ** mit folgenden Eigenschaften

- Größe (in MicroJava immer 4 Bytes)
- Struktur (Felder bei Klassen, Elementtyp bei Arrays, ...)

**Welche Typarten gibt es in MicroJava?**

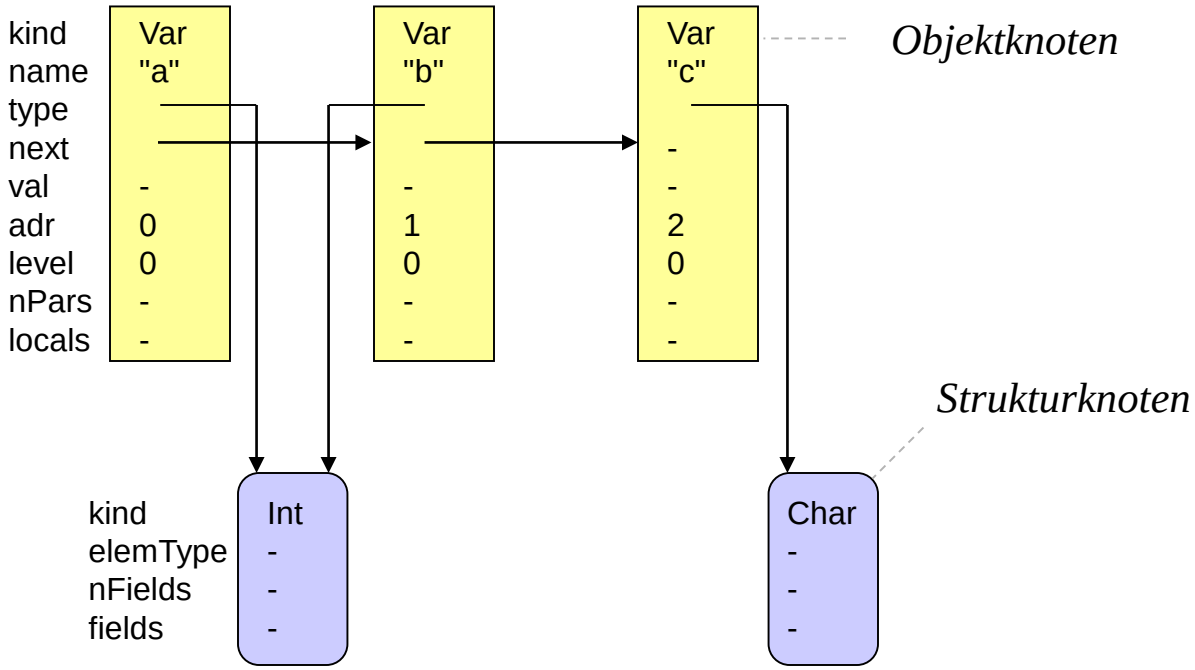
- einfache Typen (int, char)
- Arrays
- Klassen

**Typen werden durch Strukturknoten dargestellt**

```
class Struct {
   static final int          // type kinds
      None = 0, Int = 1, Char = 2, Arr = 3, Class = 4;
   int      kind;            // None, Int, Char, Arr, Class
   Struct   elemType;        // Arr: element type
   int      nFields;         // Class: number of fields
   Obj      fields;          // Class: list of fields
}
```

# *Strukturknoten für einfache Typen*

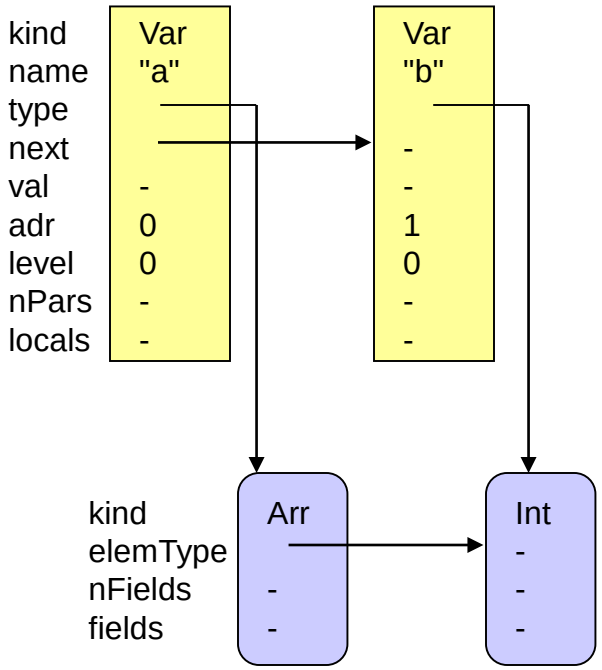
```
int a, b;
char c;
```




Es gibt in der gesamten Symbolliste nur einen einzigen Strukturknoten für *int*.
Alle Objekte vom Typ *int* verweisen auf ihn.
Dasselbe gilt für den Strukturknoten von *char* und andere Typen.

# *Strukturknoten für Arrays*

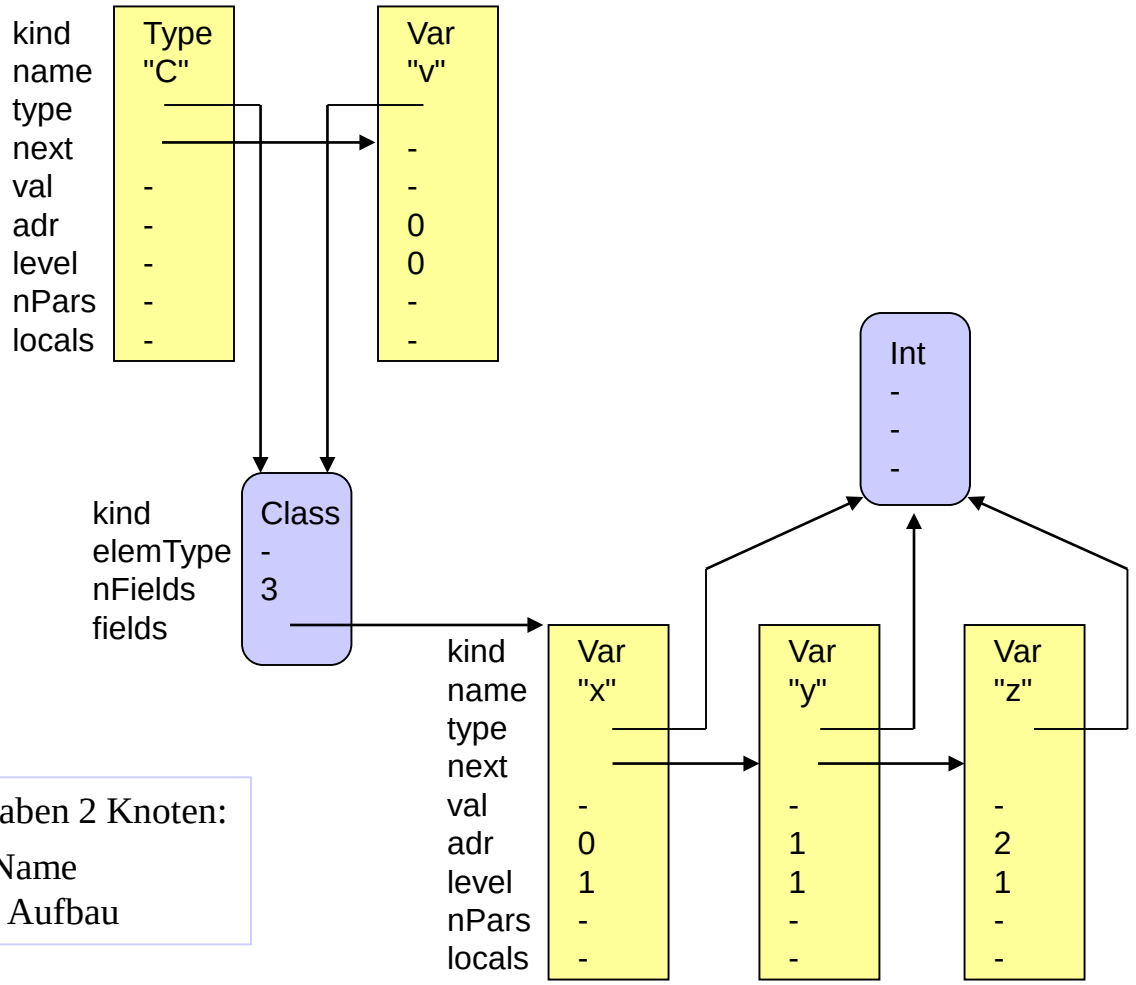
```
int[] a;
int b;
```




Die Länge eines Arrays ist statisch nicht bekannt.
Sie wird zur Laufzeit im Array-Objekt am Heap gespeichert

# *Strukturknoten für Klassen*

```
class C {
    int x;
    int y;
    int z;
}
C v;
```



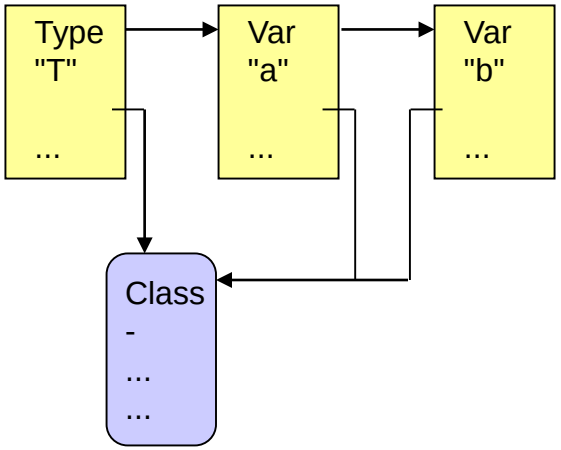

Benannte Typen haben 2 Knoten:

- Objektknoten: Name
- Strukturknoten: Aufbau

# *Typkompatibilität: Namensäquivalenz*

Typen sind gleich, wenn sie durch den gleichen Typnamen bezeichnet werden
(d.h. wenn sie durch denselben Strukturknoten dargestellt werden)

```
class T {...}
T a;
T b;
```




Die Typen von *a* und *b* sind gleich (feststellbar durch `if (a.type == b.type) ...`)

Gilt in Java, C/C++/C#, Pascal, ..., MicroJava

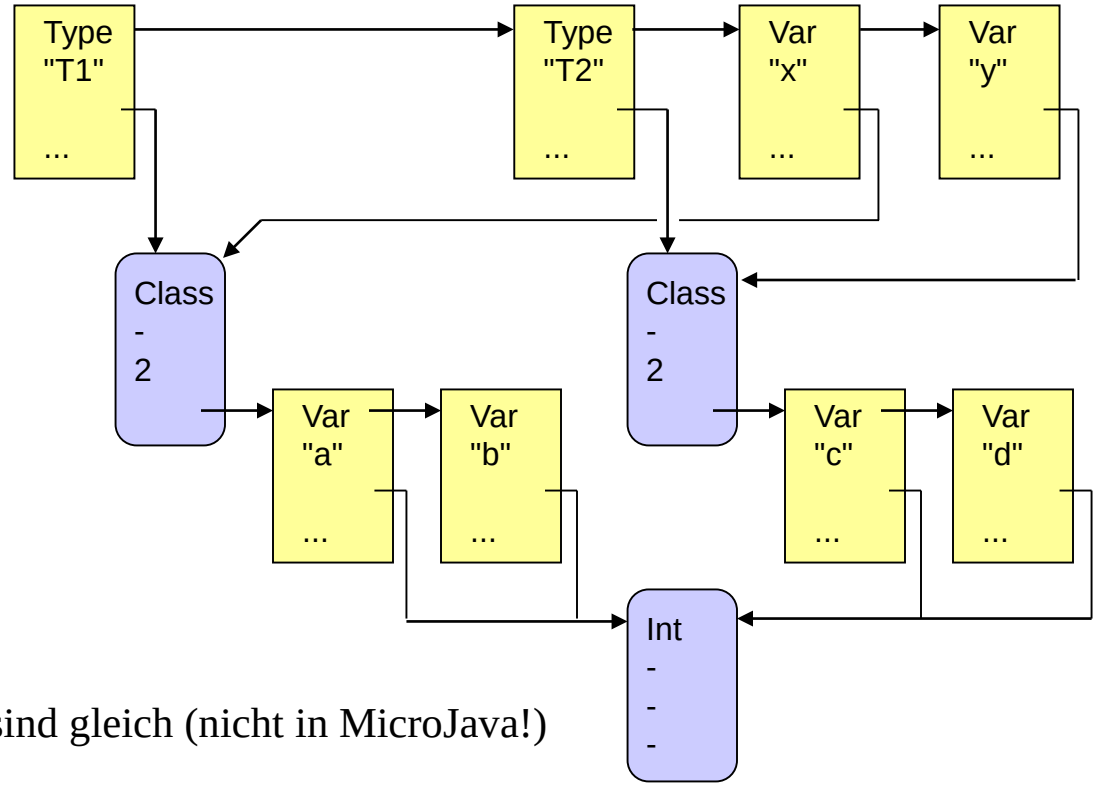
Ausnahme: Arraytypen sind in Java/MicroJava gleich, wenn sie denselben Elementtyp haben!

```
int[] a;
int[] b;
```
} gleicher Typ, obwohl nicht gleicher Typname

# *Typkompatibilität:* Strukturäquivalenz

Typen sind gleich, wenn sie die gleiche Struktur haben
(d.h. gleiche Felder vom gleichen Typ, gleichen Elementtyp, ...)

```
class T1 { int a, b; }
class T2 { int c, d; }
T1 x;
T2 y;
```




Die Typen von *x* und *y* sind gleich (nicht in MicroJava!)

Gilt z.B. in TypeScript, Modula-3 und Algol68, nicht aber in MicroJava und den meisten anderen Sprachen!

# *Methoden zur Prüfung der Typkompatibilität*


```
class Struct {
   ...
   public boolean isRefType() {
      return kind == Class || kind == Arr;
   }

   // prüft, ob zwei Typen gleich sind (bei Arrays Strukturäquivalenz, sonst Namensäquivalenz)
   public boolean equals (Struct other) {
      if (this.kind == Arr)
         return other.kind == Arr && other.elemType == this.elemType;
      else
         return other == this;
   }

   // prüft, ob this an dest zuweisbar ist
   public boolean assignableTo (Struct dest) {
      return this.equals(dest)
         ||  this == Tab.nullType && dest.isRefType()
         ||  this.kind == Arr && dest.kind == Arr && dest.elemType == Tab.noType;
   }                                                  nötig wegen Standardfunktion len(arr)

   // prüft, ob zwei Typen kompatibel sind (z.B. in Vergleichen)
   public boolean compatibleWith (Struct other) {
      return this.equals(other)
         ||  this == Tab.nullType && other.isRefType()
         ||  other == Tab.nullType && this.isRefType();
   }
}
```

# *Lösen von LL(1)-Konflikten mittels Symbolliste*


*Methodensyntax in MicroJava*

```
void foo()
   int a;
{ a = 0; ...
}
```

*Besser wäre eigentlich*

```
void foo() {
   int a;
   a = 0; ...
}
```

*Das ergäbe aber einen LL(1)-Konflikt*

First(*VarDecl*) ∩ First(*Statement*) = {*ident*}

```
Block       =  "{" { VarDecl | Statement } "}".
VarDecl     =  Type ident {"," ident}.
Type        =  ident ["[" "]"].
Statement   =  Designator "=" Expr ";"
            |  ... .
Designator  =  ident {"." ident | "[" Expr "]"}.
```

Syntaktische Auflösung dieses Konflikts wäre sehr umständlich.

# *Lösen des Konflikts mit semantischer Information*


```
private static void Block() {
   check(lbrace);
   for (;;) {
      if (nextTokenIsType()) VarDecl();
      else if (sym ∈ First(Statement)) Statement();
      else if (sym ∈ {rbrace, eof}) break;
      else {
         error("..."); ... recover ...
      }
   }
   check(rbrace);
}
```

Block = "{" { VarDecl | Statement } "}".

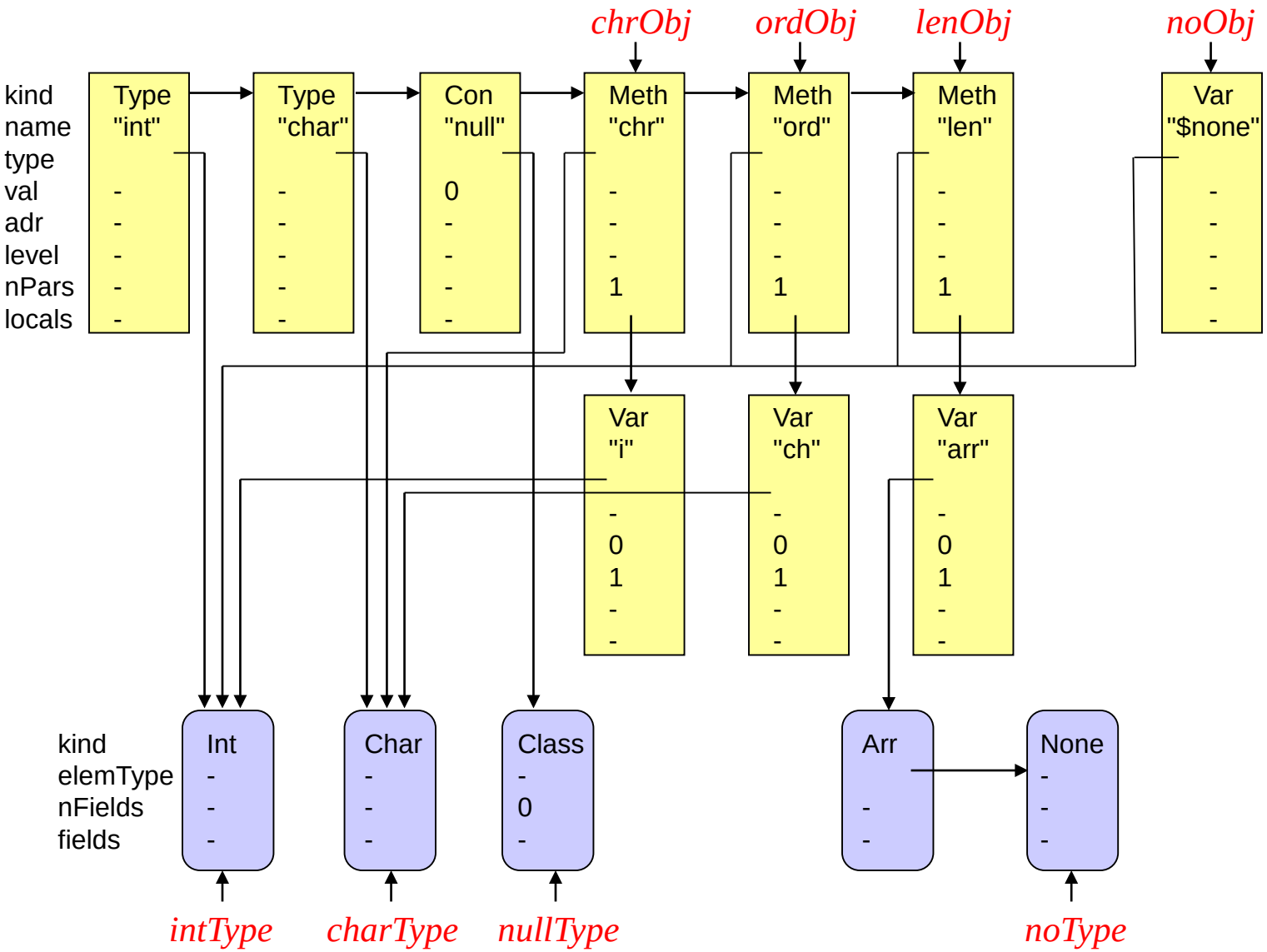
- *VarDecl* beginnt mit einem Typnamen
- *Statement* beginnt mit einem Variablen- oder Methodennamen

```
private static boolean nextTokenIsType() {
   if (sym != ident) return false;
   Obj obj = Tab.find(la.val);
   return obj.kind == Obj.Type;
}
```

prüft, ob das nächste Symbol ein Typname ist


# 5. Symbolliste

# Aufbau des "Universums"



# *Schnittstelle der Symbolliste*


```
class Tab {
   static Scope  curScope;    // current scope
   static int    curLevel;    // nesting level of current scope

   static Struct intType;     // predeclared types
   static Struct charType;
   static Struct nullType;
   static Struct noType;

   static Obj    chrObj;      // predeclared objects
   static Obj    ordObj;
   static Obj    lenObj;
   static Obj    noObj;

   static Obj    insert (int kind, String name, Struct type) {...}
   static Obj    find (String name) {...}
   static void   openScope() {...}
   static void   closeScope() {...}

   static void   init() {...}     // builds the universe and initializes Tab
}
```