5. Symbol Table

Responsibilities of the Symbol Table

1. It maintains all declared names and their properties

- type
- value (for named constants)
- address (for variables, fields and methods)
- parameters (for methods)
- ...

2. It is used to retrieve the properties of a name

- Mapping: name ⇒ (type, value, address, ...)

3. It manages the scopes of names

Contents of the symbol table

- *Object* nodes: Information about declared names
- *Structure* nodes: Information about type structures
- *Scope* nodes: for managing the visibility of names

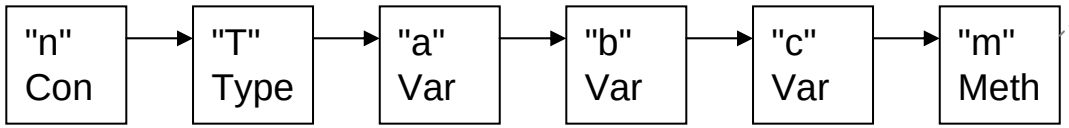
=> most suitably implemented as a dynamic data structure
(linear list, binary tree, hash table)

Symbol Table as a Linear List

Given the following declarations

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

we get the following linear list



for every declared name there is an Object node

\+ simple
\+ declaration order is retained (important if addresses are assigned only later)

\- slow if there are many declarations

Basic interface

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

5. Symbol Table

Object Nodes

Every declared name is stored in an object node

Kinds of objects in MicroJava

- constants
- variables and fields
- types
- methods

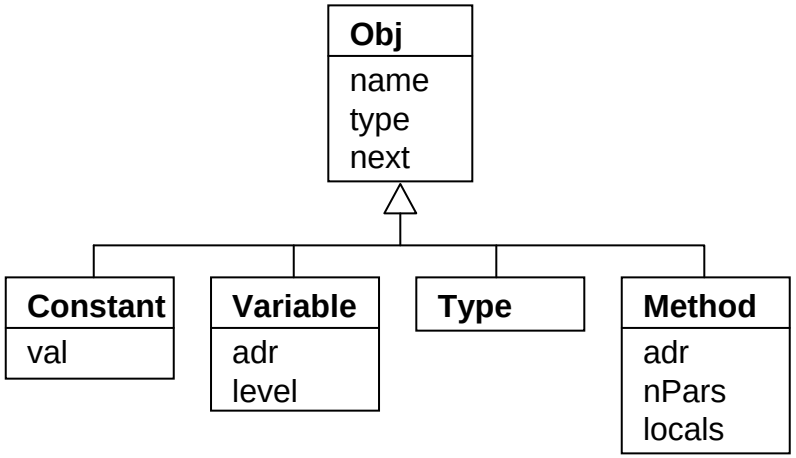
```
static final int
   Con   = 0,
   Var   = 1,
   Type  = 2,
   Meth  = 3;
```

What information is needed about objects?

- for all objects — name, type structure, object kind, pointer to the next object
- for constants — value
- for variables — address, declaration level
- for types — -
- for methods — address, number of parameters, local objects

Possible Object-oriented Architecture

Possible class hierarchy of objects



However, this is too complicated because it would require too many type casts

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

Therefore we choose a "flat implementation": all information is stored in a single class.
This is ok because

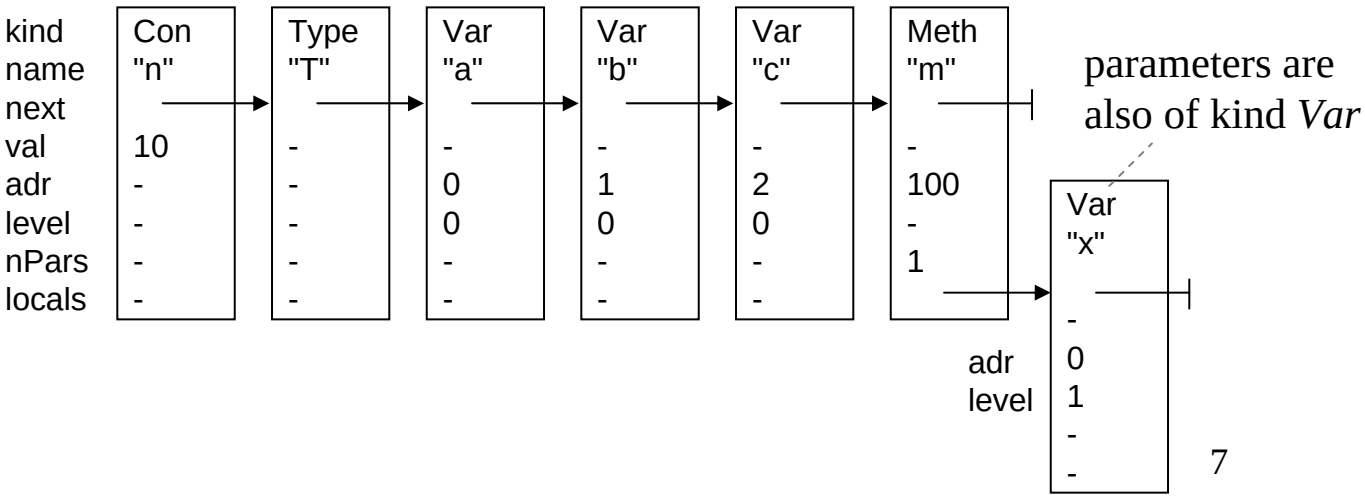
- extensibility is not required: we never need to add new object variants
- we do not need dynamically bound method calls

Class 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
}
```

Example

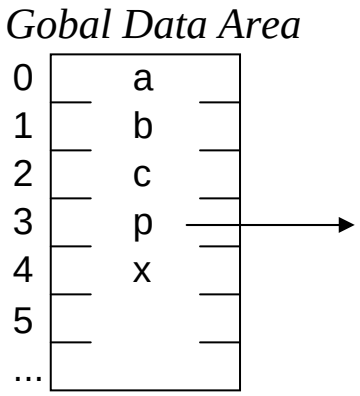
```
final int n = 10;
class T { ... }
int a, b, c;
void m(int x) { ... }
```



Global Variables

Global variables are stored in the *Global Data Area* of the MicroJava VM

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

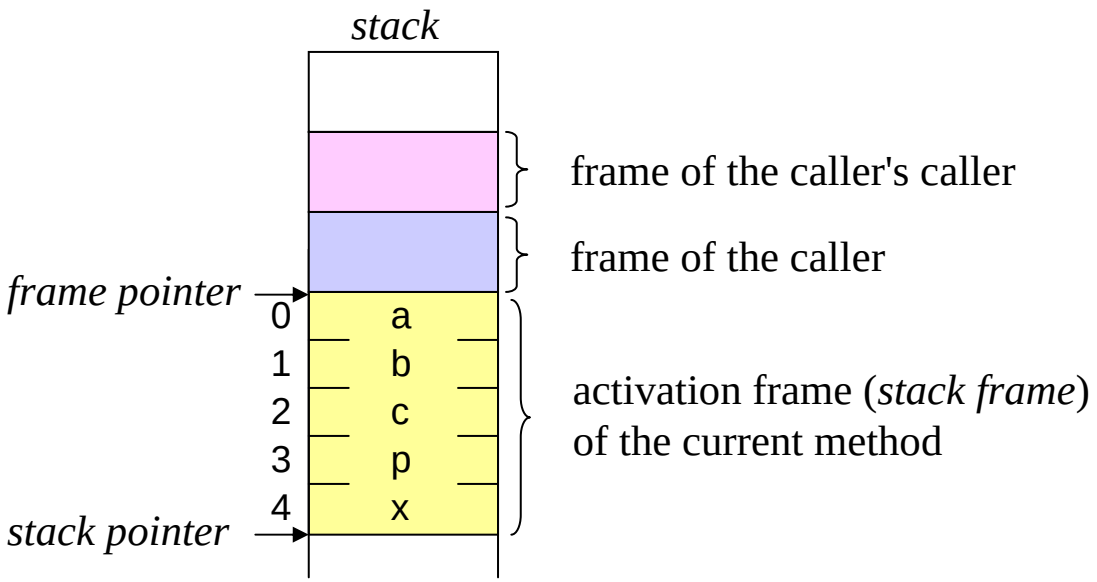


- Every variable occupies 1 word (4 bytes)
- Addresses are word numbers relative to the Global Data Area
- Addresses are allocated sequentially in the order of declaration

Local Variables

Local variables are stored in an "activation frame" on the method stack

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



- Every variable occupies 1 word (4 bytes)
- Addresses are word numbers relative to the *frame pointer*
- Addresses are allocated sequentially in the order of declaration

Entering Names into the Symbol Table

The following method is called whenever a name is declared

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

- creates a new object node with *kind*, *name*, *type*
- checks if *name* is already declared (if so => error message)
- assigns successive addresses to variables and fields
- enters the declaration level for variables (0 = global, 1 = local)
- appends the new node to the end of the symbol table
- returns the new node to the caller

Example for calling *insert()*

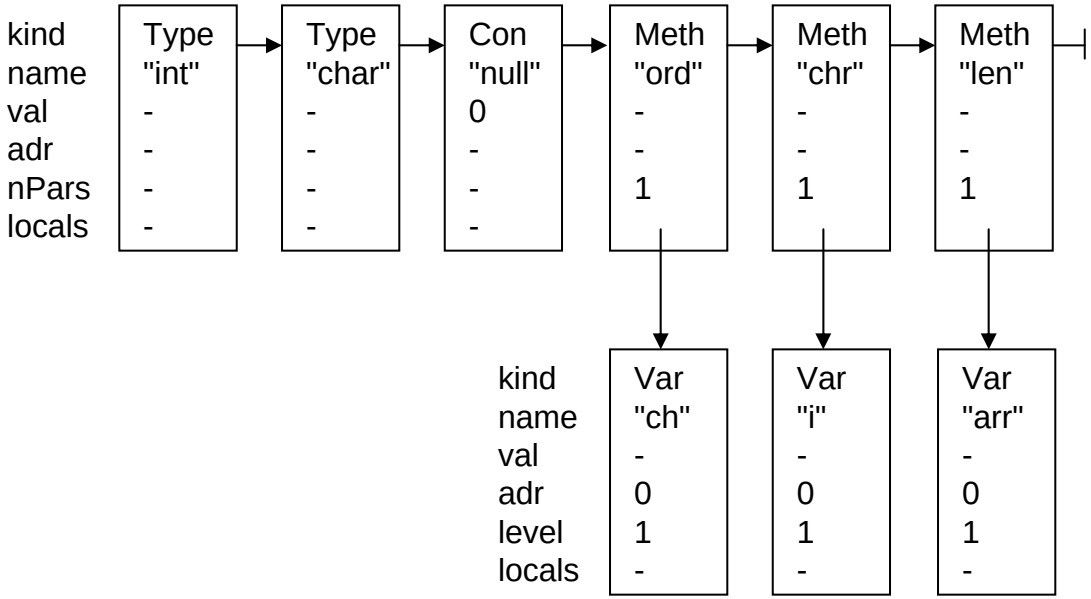
```
VarDecl
=  Type<↑type>
   ident<↑name>          (. Tab.insert(Obj.Var, name, type); .)
   { "," ident<↑name>    (. Tab.insert(Obj.Var, name, type); .)
   }
   ";" .
```

Predeclared Names

Which names are predeclared in MicroJava?

- Standard types: int, char
- Standard constants: null
- Standard methods: ord(ch), chr(i), len(arr)

Predeclared names are also stored in the symbol table



Special Names as Keywords

int and _char_ could also be implemented as keywords

requires a special treatment in the grammar

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

It is simpler to have them predeclared in the symbol table

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

\+ uniform treatment of predeclared and user-declared names
\- one can redeclare "int" as a user type

5. Symbol Table

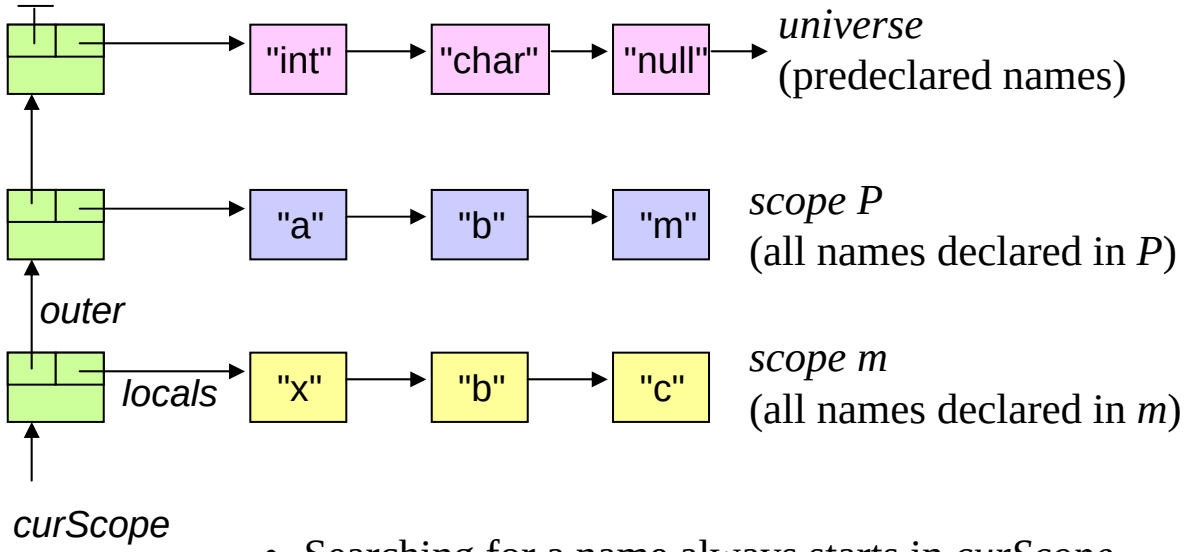
Scope = Range in which a Name is Valid

There are separate scopes (object lists) for

- the program — contains global names
- every method — contains local names
- every class — contains fields
- the "universe" — contains the predeclared names

Example

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



- Searching for a name always starts in *curScope*
- If not found, the search continues in the next outer scope
- Example: search *b*, *a* and *null*

Scope Nodes


```
class Scope {
   Scope  outer;    // to the next outer scope
   Obj    locals;   // to the objects in this scope
   int    nVars;    // number of variables in this scope (for address allocation)
}
```

Method for opening a scope

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

- called at the beginning of a method or class
- links the new scope with the existing ones
- new scope becomes *curScope*
- *Tab.insert()* always creates objects in *curScope*

Method for closing a scope

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

- called at the end of a method or class
- next outer scope becomes *curScope*

Entering Names into a Scope

Names are always entered in *curScope*

```
class Tab {
   static Scope curScope;   // current scope
   static int curLevel;     // current declaration level (0 = global, 1 = local)
   ...
   static Obj insert (int kind, String name, Struct type) {
      //--- create object node
      Obj obj = new Obj(kind, name, type);
      if (kind == Obj.Var) {
         obj.adr = curScope.nVars; curScope.nVars++;
         obj.level = curLevel;
      }
      //--- append object node
      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;
   }
   ...
}
```

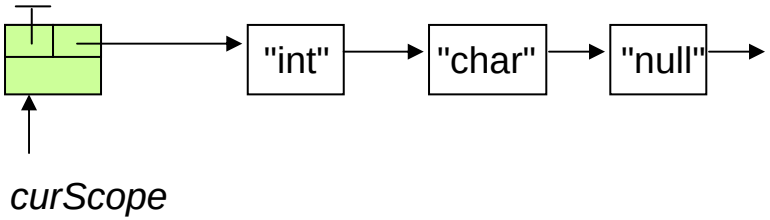
Opening and Closing a Scope


```
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(); .)
   .
```

Note

- The method name is entered in the method's enclosing scope
- *curMethod* is a global variable of type *Obj*
- Before a scope is closed its local objects are assigned to *curMethod.locals*
- Scopes are also opened and closed for classes

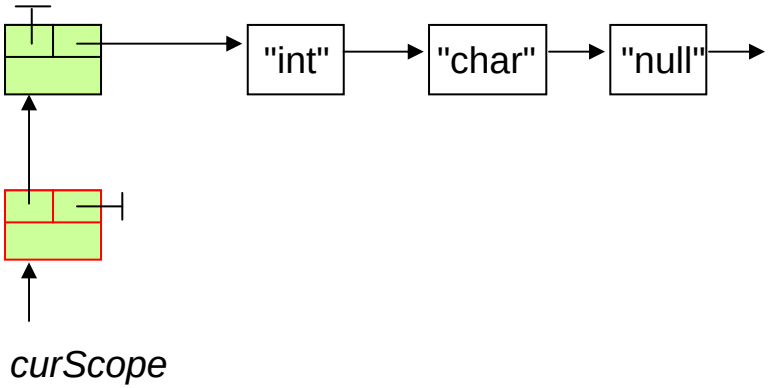
Example



Example

program P

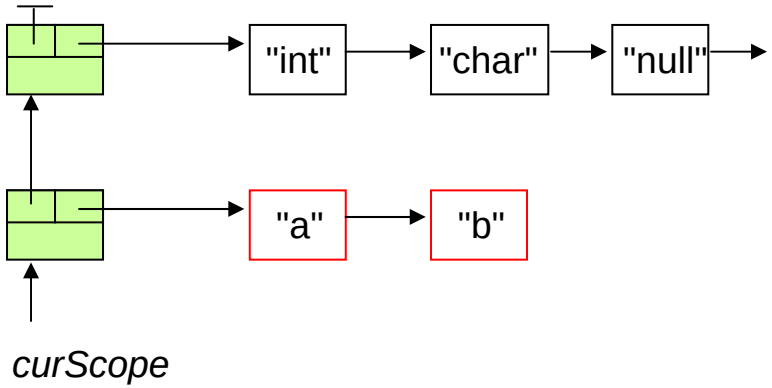
Tab.openScope();



Example

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

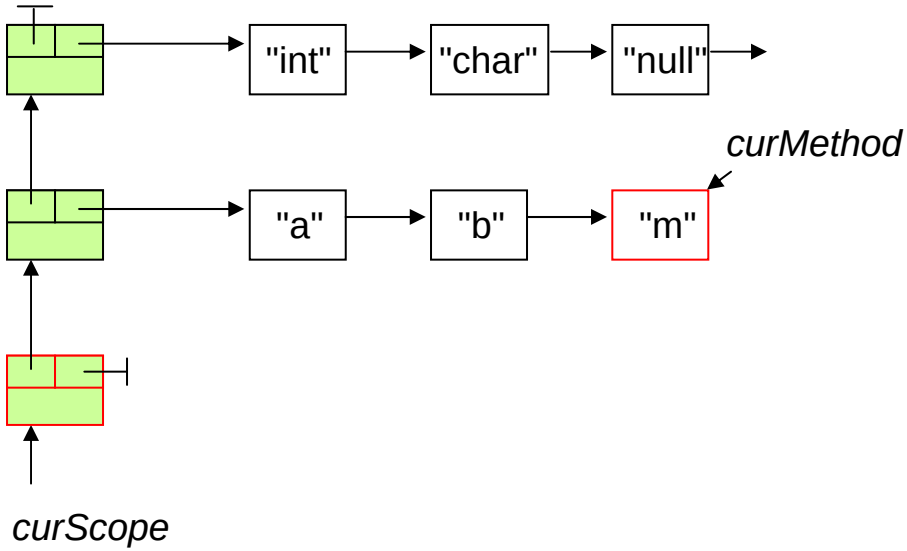
```
Tab.insert(..., "a", ...);
Tab.insert(..., "b", ...);
```



Example

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

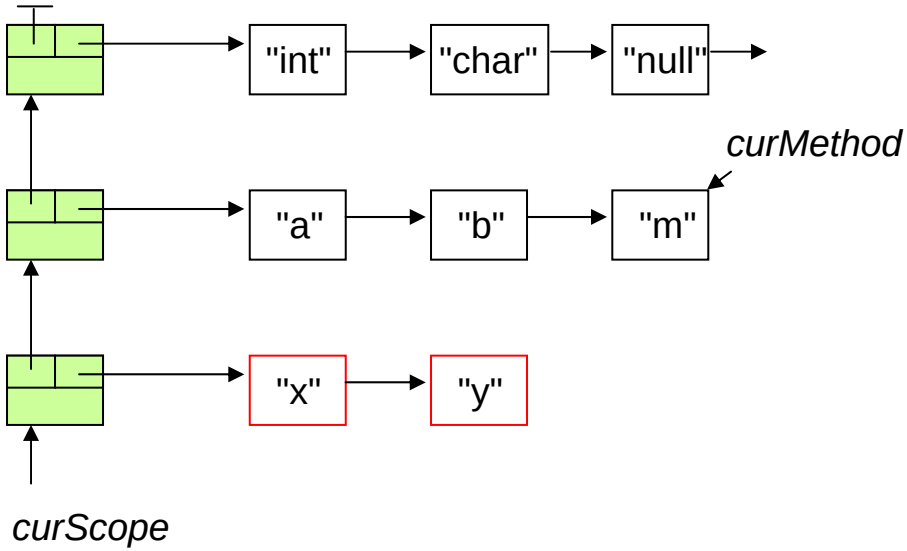
```
Tab.insert(..., "m", ...);
Tab.openScope();
```



Example

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

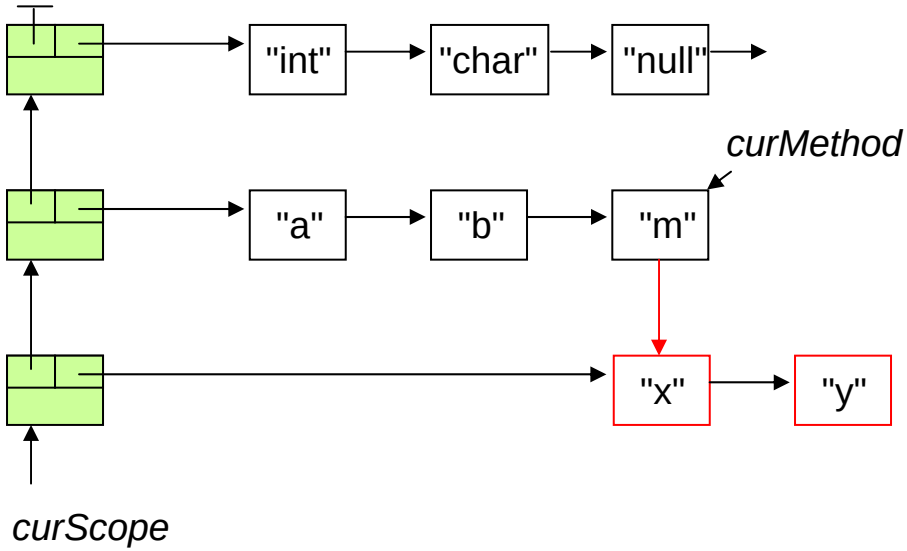
```
Tab.insert(..., "x", ...);
Tab.insert(..., "y", ...);
```



Example

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

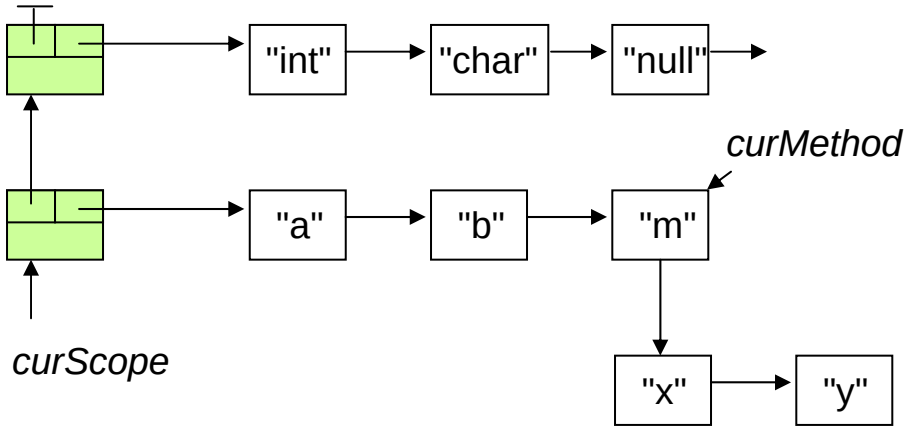
```
curMethod.locals =
    Tab.curScope.locals;
```



Example

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

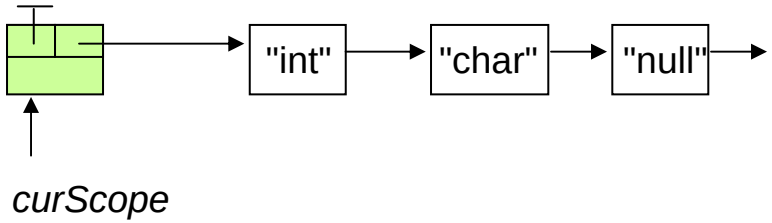
Tab.closeScope();



Example

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

Tab.closeScope();



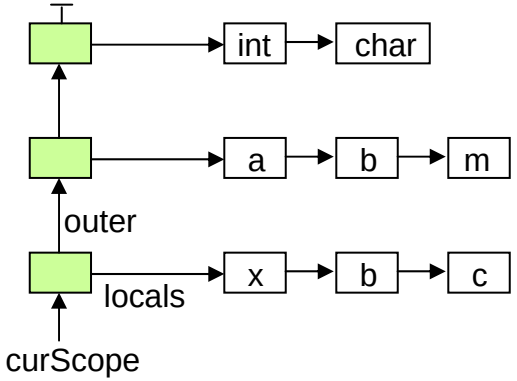
Searching Names in the Symbol Table

The following method is called whenever a name is used

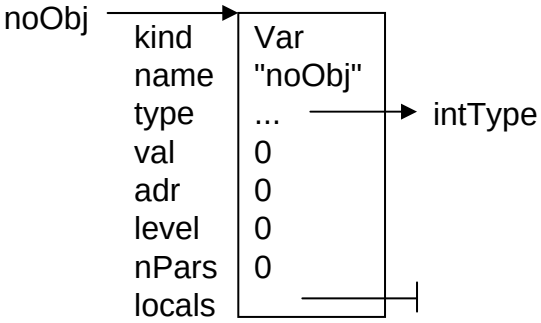
```
Obj obj = Tab.find(name);
```

- The lookup starts in *curScope*
- If not found, the lookup is continued in the next outer scope

```
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;
}
```



If a name is not found the method returns *noObj*



- predeclared dummy object
- better than *null*, because it avoids aftereffects (exceptions)

5. Symbol Table

Types

Every object has a type with the following properties

- size (in MicroJava always 4 bytes)
- structure (fields for classes, element type for arrays, ...)

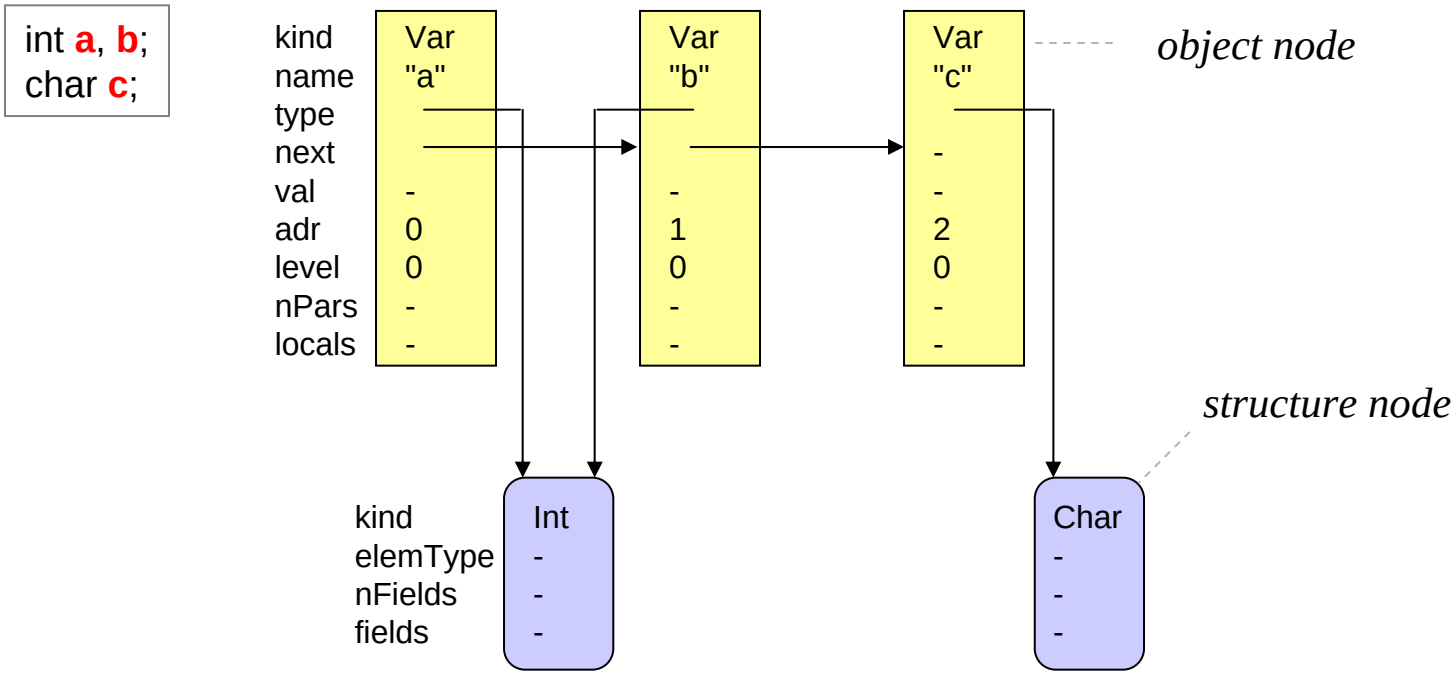
Kinds of types in MicroJava?

- primitive types (int, char)
- arrays
- classes

Types are represented by structure nodes

```
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
}
```

Structure Nodes for Primitive Types

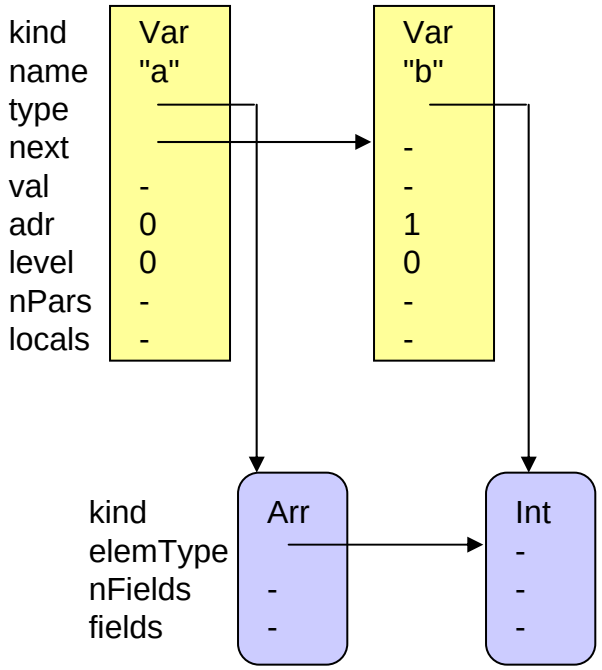


There is just a single structure node for *int* in the whole symbol table. It is referenced by all objects of type *int*.

The same is true for structure nodes of kind *char*.

Structure Nodes for Arrays

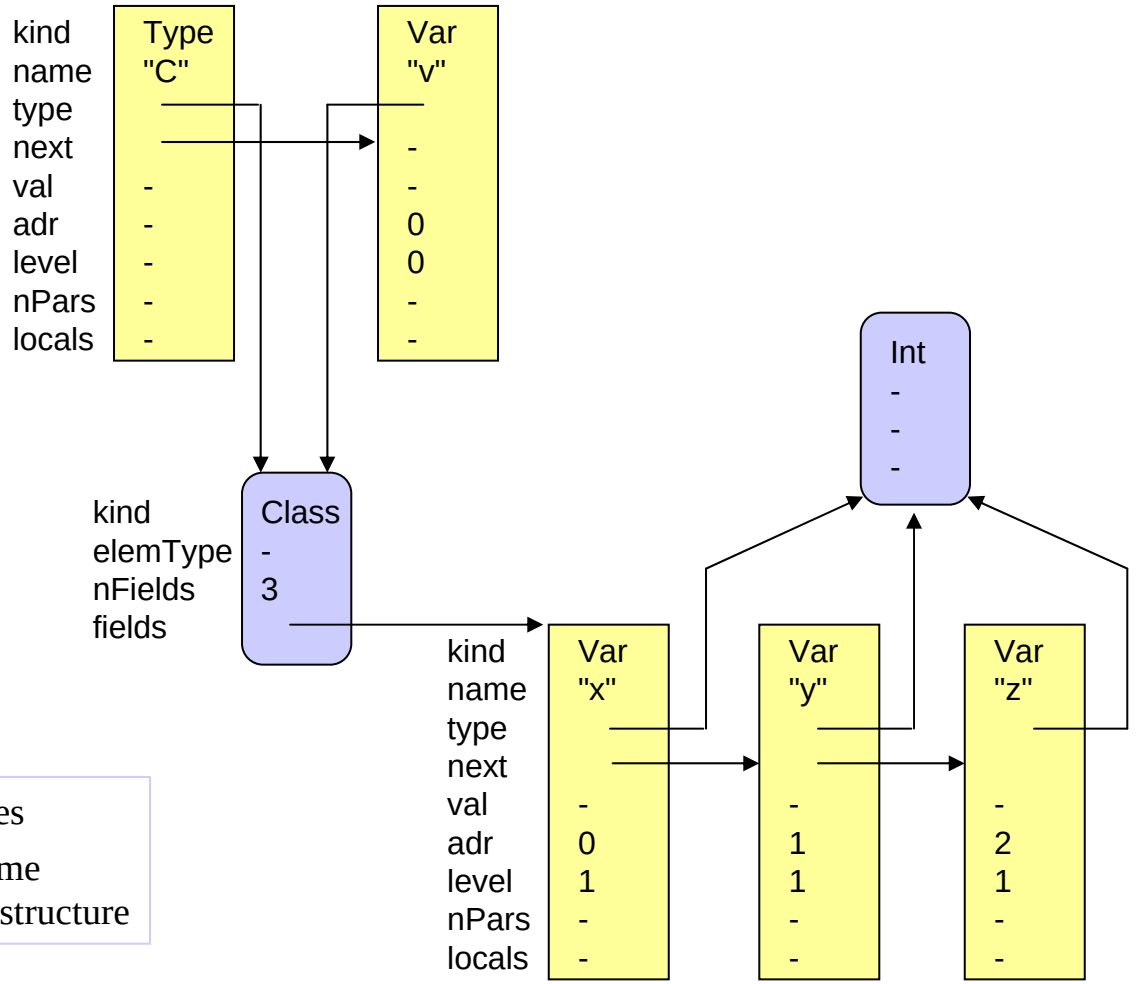
```
int[] a;
int b;
```



The length of an array is statically unknown.
It is stored in the array at run time.

Structure Nodes for Classes

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



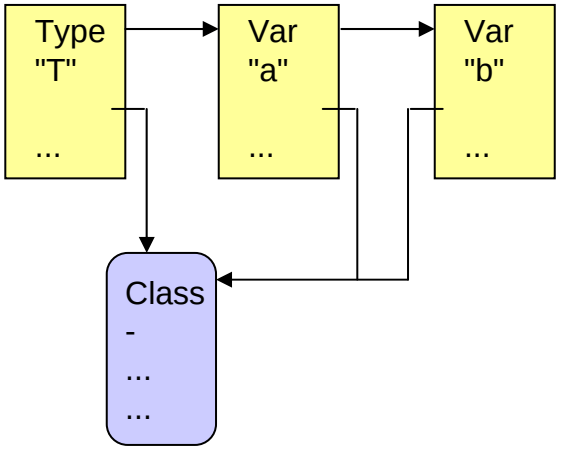
Types have 2 nodes

- object node: name
- structure node: structure

Type Compatibility: Name Equivalence

Two types are the same if they are denoted by the same name
(i.e. if they are represented by the same type node)

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



The types of *a* and *b* are the same (can be checked by if (a.type == b.type) ...)

Name equivalence is used in Java, C/C++/C#, Pascal, ..., MicroJava

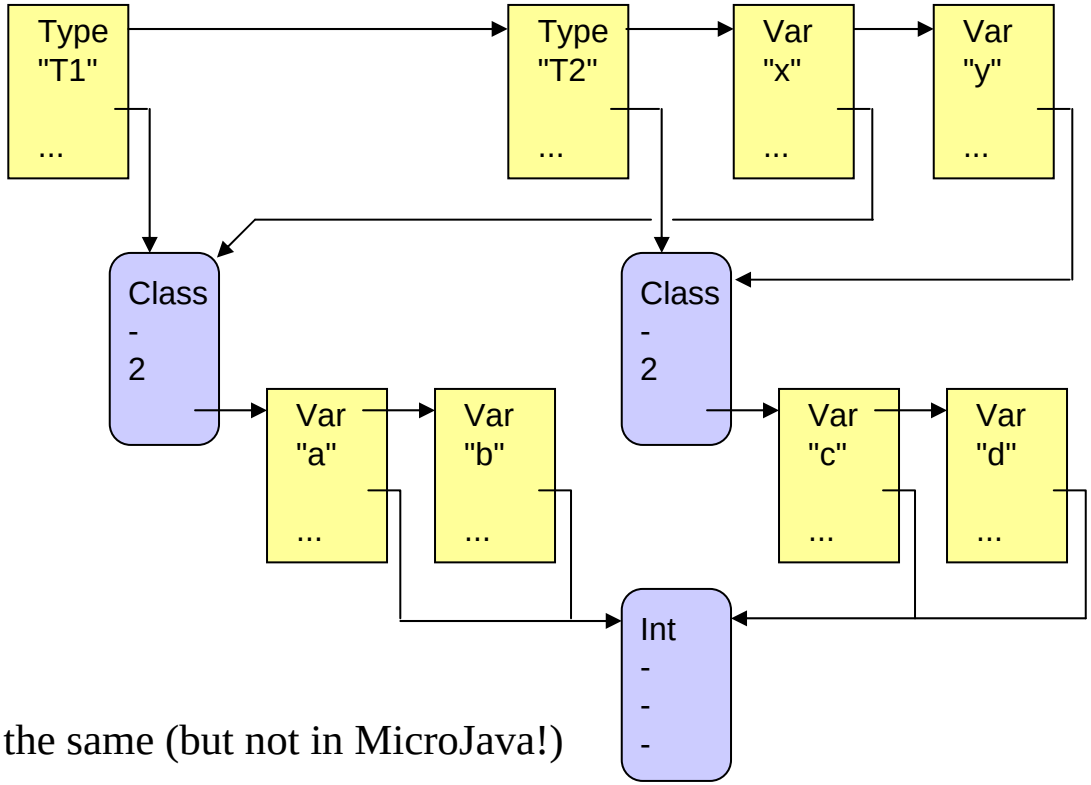
Exception
In Java (and MicroJava) two array types are the same if they have the same element types!

```
int[] a;
int[] b;
```
} same types although different type names

Type Compatibility: Structural Equivalence

Two types are the same if they have the same structure
(i.e. the same fields of the same types, the same element type, ...)

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



The types of *x* and *y* are the same (but not in MicroJava!)

Structural equivalence is used in Modula-3 but not in MicroJava and in most other languages!

Methods for Checking Type Compatibility


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

   // checks if two types are the same (structural equivalence for arrays, name equivalence otherwise)
   public boolean equals (Struct other) {
      if (this.kind == Arr)
         return other.kind == Arr && other.elemType == this.elemType;
      else
         return other == this;
   }

   // checks if "this" is assignable to "dest"
   public boolean assignableTo (Struct dest) {
      return this.equals(dest)
         ||  this == Tab.nullType && dest.isRefType()
         ||  this.kind == Arr && dest.kind == Arr && dest.elemType = Tab.noType;
   }                                                  necessary because of standard function len(arr)

   // checks if two types are compatible (e.g. in compare operations)
   public boolean compatibleWith (Struct other) {
      return this.equals(other)
         ||  this == Tab.nullType && other.isRefType()
         ||  other == Tab.nullType && this.isRefType();
   }
}
```

Solving LL(1) Conflicts with the Symbol Table

Method syntax in MicroJava

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

Actually we would like to write it like this

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

But this would result in an LL(1) conflict

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

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

Solving the Conflict With Semantic 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);
}
```

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

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

5. Symbol Table

5.1 Overview

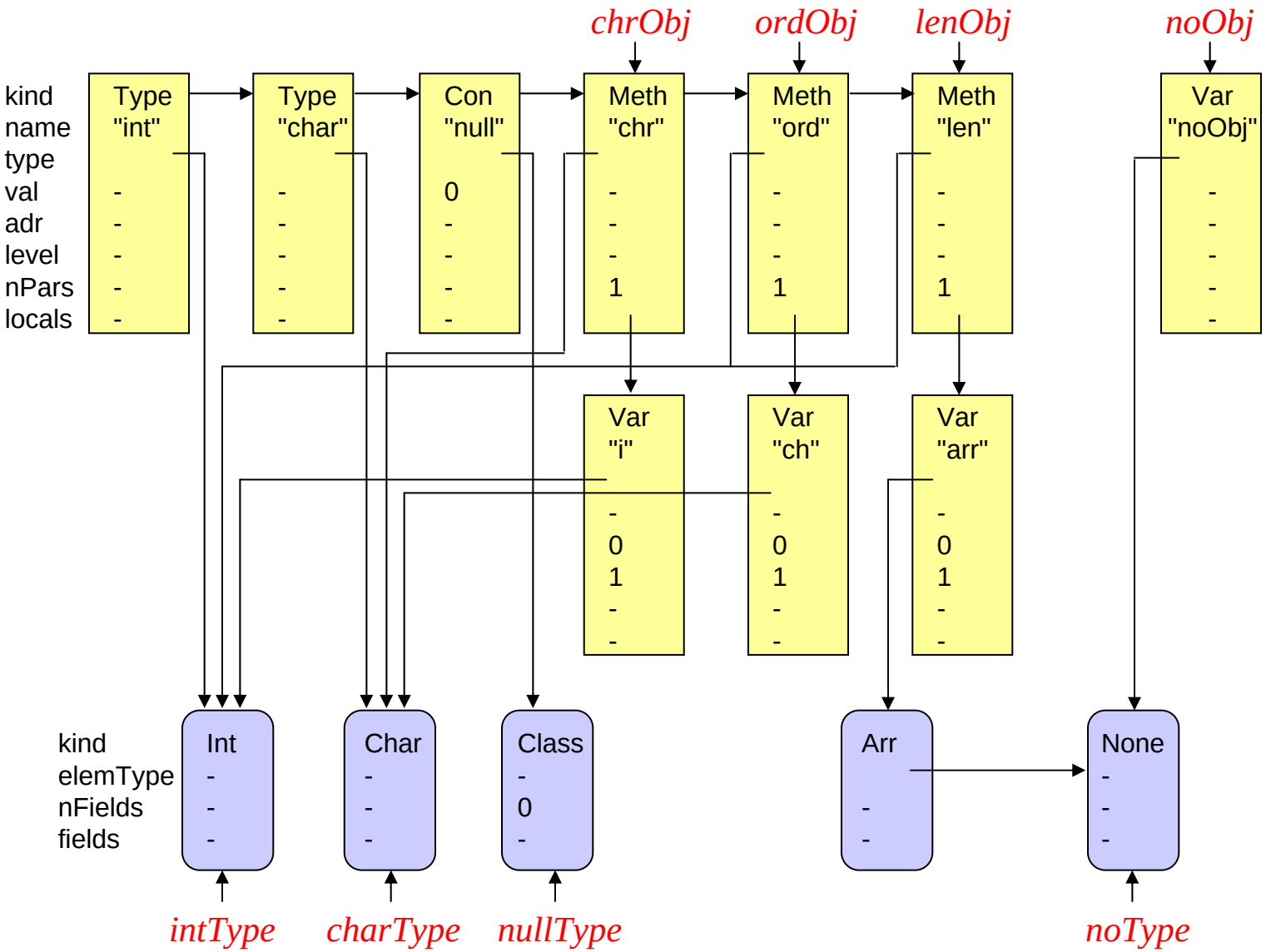
5.2 Objects

5.3 Scopes

5.4 Types

5.5 Universe

Structure of the "universe"



Interface of the Symbol Table


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

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

   static Obj    chrObj;     // predefined 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
}
```