5. Symbol Table

Responsibilities of the Symbol Table

1. It stores all declared names and their attributes

- type
- value (for constants)
- address (for local variables and method arguments)
- parameters (for methods)
- ...

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

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

Contents of the symbol table

- *Symbol* nodes: information about declared names
- *Structure* nodes: information about type structures

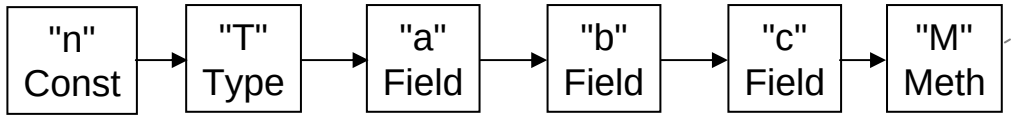
=> most suitably implemented as a dynamic data structure
- linear list
- binary tree
- hash table

Symbol Table as a Linear List

Given the following declarations

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

we get the following linear list



for every declared name there is a Symbol node

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

\- slow if there are many declarations

Basic interface

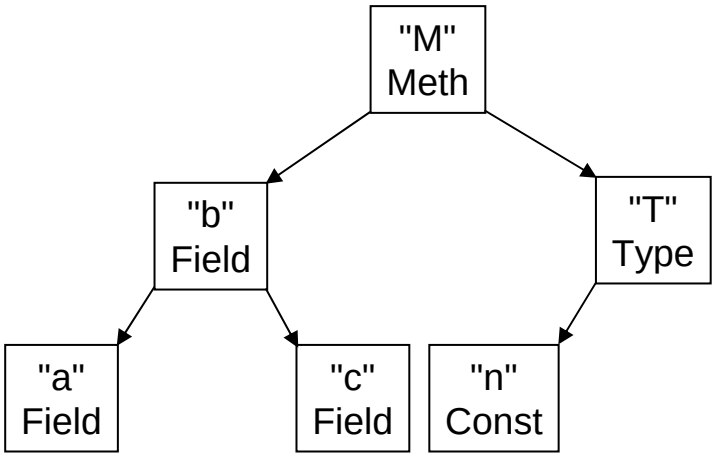
```
public class Tab {
   public static Symbol Insert (Symbol.Kinds kind, string name, ...);
   public static Symbol Find (string name);
}
```

Symbol Table as a Binary Tree

Declarations

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

Resulting binary tree



\+ fast

\- can degenerate unless it is balanced
\- larger memory consumption
\- declaration order is lost

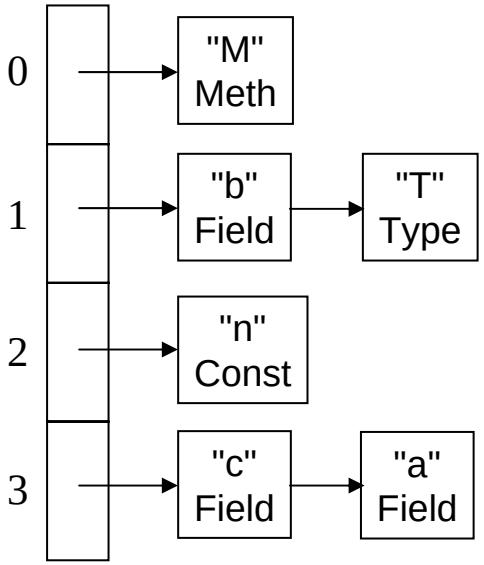
Only useful if there are many declarations

Symbol Table as a Hashtable

Declarations

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

Resulting hashtable



\+ fast

\- more complicated than a linear list
\- declaration order is lost

For our purposes a linear list is sufficient

- Every scope is a list of its own anyway
- A scope has hardly more than 10 names

5. Symbol Table

Symbol Nodes

Every declared name is stored in a Symbol node

Kinds of symbols in Z#

- constants
- global variables
- fields
- method arguments
- local variables
- types
- methods
- program

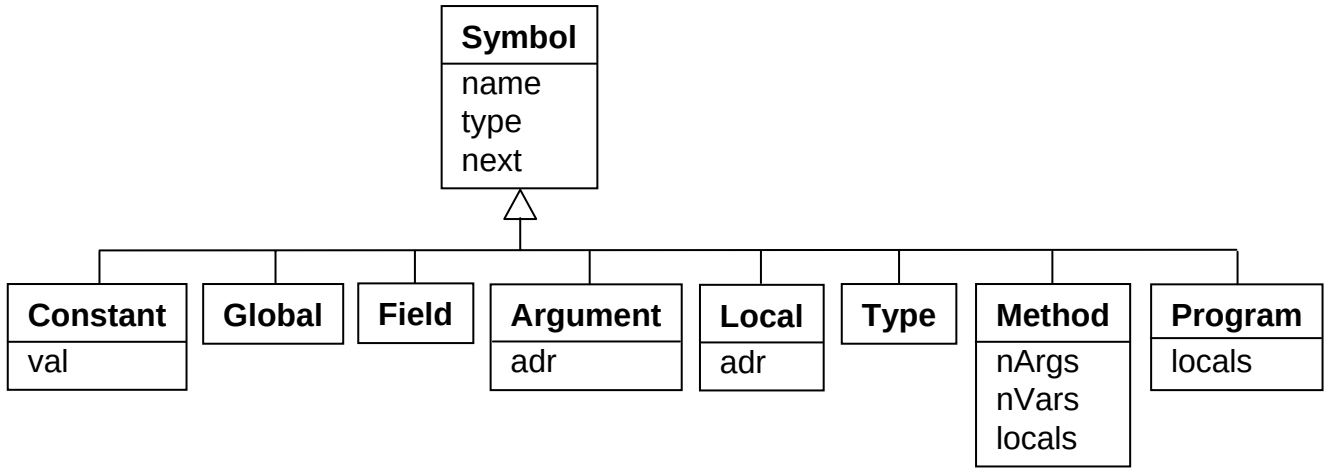
```
public enum Kinds {
   Const,
   Global,
   Field,
   Arg,
   Local,
   Type,
   Meth,
   Prog
}
```

What information is needed about objects?

• for all symbols	name, type structure, symbol kind, pointer to the next symbol
• for constants	value
• for method arguments	address (= order of declaration)
• for local variables	address (= order of declaration)
• for methods	number of arguments and local variables, local symbols (args + local vars)
• for program	global symbols (= local to the program)
• for global vars, fields, types	---

Possible Object-oriented Architecture

Possible class hierarchy of objects



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

```
Symbol sym = Tab.Find("x");
if (sym is Argument) ((Argument) sym).adr = ...;
else if (sym is Method) ((Method) sym).nArgs = ...;
...
```

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

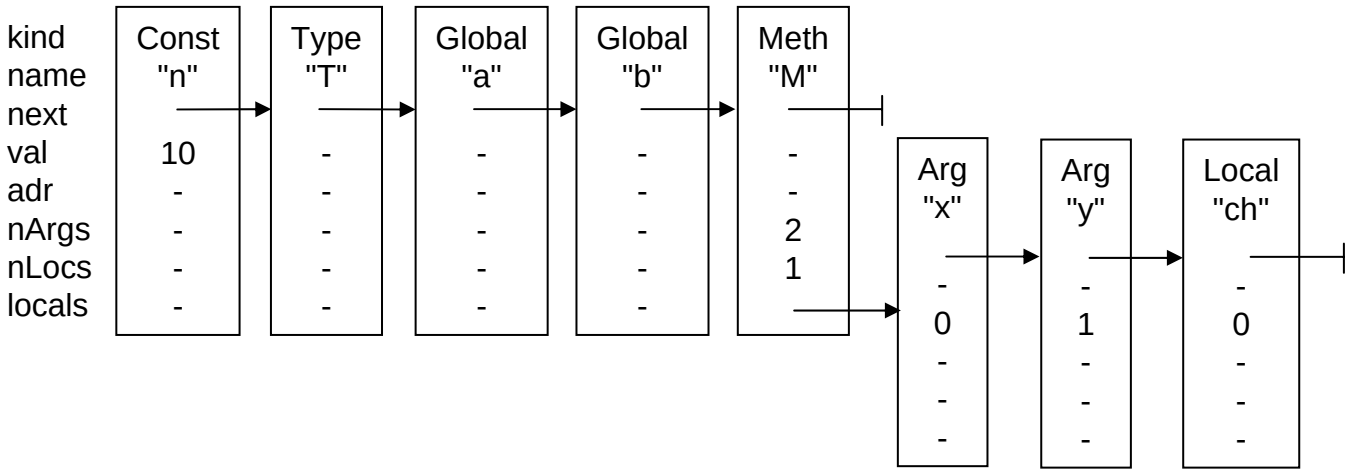
- extensibility is not required: we never need to add new object variants
- we do not need dynamically bound method calls

Class Symbol


```
class Symbol {
   public enum Kinds { Const, Global, Field, Arg, Local, Type, Meth, Prog }
   Kinds    kind;
   string   name;
   Struct   type;
   Symbol   next;
   int      val;      // Const: value
   int      adr;      // Arg, Local: address
   int      nArgs;    // Meth: number of arguments
   int      nLocs;    // Meth: number of local variables
   Symbol   locals;   // Meth: parameters & local variables; Prog: symbol table of program
}
```

Example

```
const int n = 10;
class T { ... }
int a, b;
void M (int x, int y)
   char ch;
{ ... }
```



Entering Names into the Symbol Table

The following method is called whenever a name is declared

```
Symbol sym = 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 list
- returns the new node to the caller

Example for calling *Insert()*

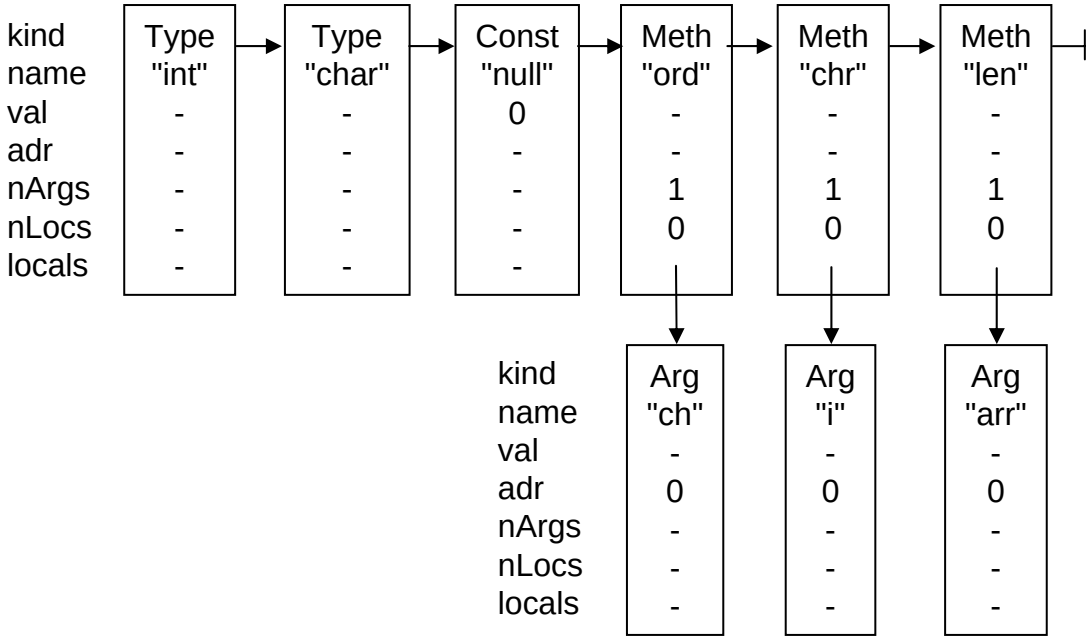
```
VarDecl<↓Symbol.Kinds kind>
=  Type<↑type>
   ident                    (. Tab.insert(Obj.Var, name, type); .)
   { ";" ident              (. Tab.insert(Obj.Var, name, type); .)
   }.
```

Predeclared Names

Which names are predeclared in Z#?

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

Predeclared names are also stored in the symbol table ("Universe")



Special Names as Keywords

***int* and *char* could also be implemented as keywords.**

requires a special treatment in the grammar

```
Type<↑Struct type>
=  ident                (. Symbol sym = Tab.Find(token.str); type = sym.type; .)
|  "int"                (. type = Tab.intType; .)
|  "char"               (. type = Tab.charType; .)
.
```

It is simpler to have them predeclared in the symbol table.

```
Type<↑Struct type>
=  ident                (. Symbol sym = Tab.Find(token.str); type = sym.type; .)
```

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

5. Symbol Table

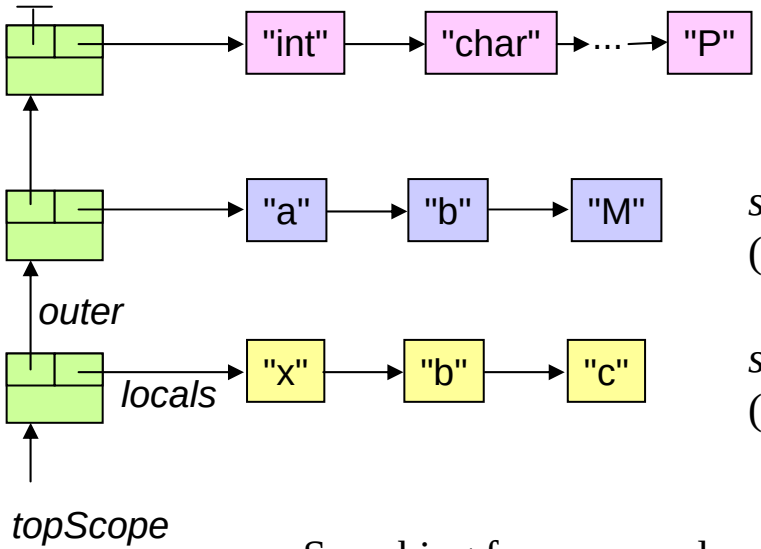
Scope = Range in which a Name is Valid

There are separate scopes (object lists) for

- the "universe" contains the predeclared names (and the program symbol)
- the program contains global names (= constants, global variables, classes, methods)
- every method contains local names (= argument and local variables)
- every class contains fields

Example

```
class P
   int a, b;
{
   void M (int x)
      int b, c;
   {
      ...
   }
   ...
}
```



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

Scope Nodes


```
class Scope {
   Scope   outer;   // to the next outer scope
   Symbol  locals;  // to the symbols in this scope
   int     nArgs;   // number of arguments in this scope (for address allocation)
   int     nLocs;   // number of local variables in this scope (for address allocation)
}
```

Method for opening a scope

```
static void OpenScope () { // in class Tab
   Scope s = new Scope();
   s.nArgs = 0; s.nLocs = 0;
   s.outer = topScope;
   topScope = s;
}
```

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

Method for closing a scope

```
static void CloseScope () { // in class Tab
   topScope = topScope.outer;
}
```

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

Entering Names in Scope

Names are always entered in *topScope*

```
class Tab {
   Scope topScope;  // Zeiger auf aktuellen Scope
   ...
   static Symbol Insert (Symbol.Kinds kind, string name, Struct type) {
      //--- create symbol node
      Symbol sym = new Symbol(name, kind, type);

      if (kind == Symbol.Kinds.Arg) sym.adr = topScope.nArgs++;
      else if (kind == Symbol.Kinds.Local) sym.adr = topScope.nLocs++;

      //--- insert symbol node
      Symbol cur = topScope.locals, last = null;
      while (cur != null) {
         if (cur.name == name) Error(name + " declared twice");
         last = cur; cur = cur.next;
      }
      if (last == null) topScope.locals = sym;
      else last.next = sym;
      return sym;
   }
   ...
}
```

Opening and Closing a Scope

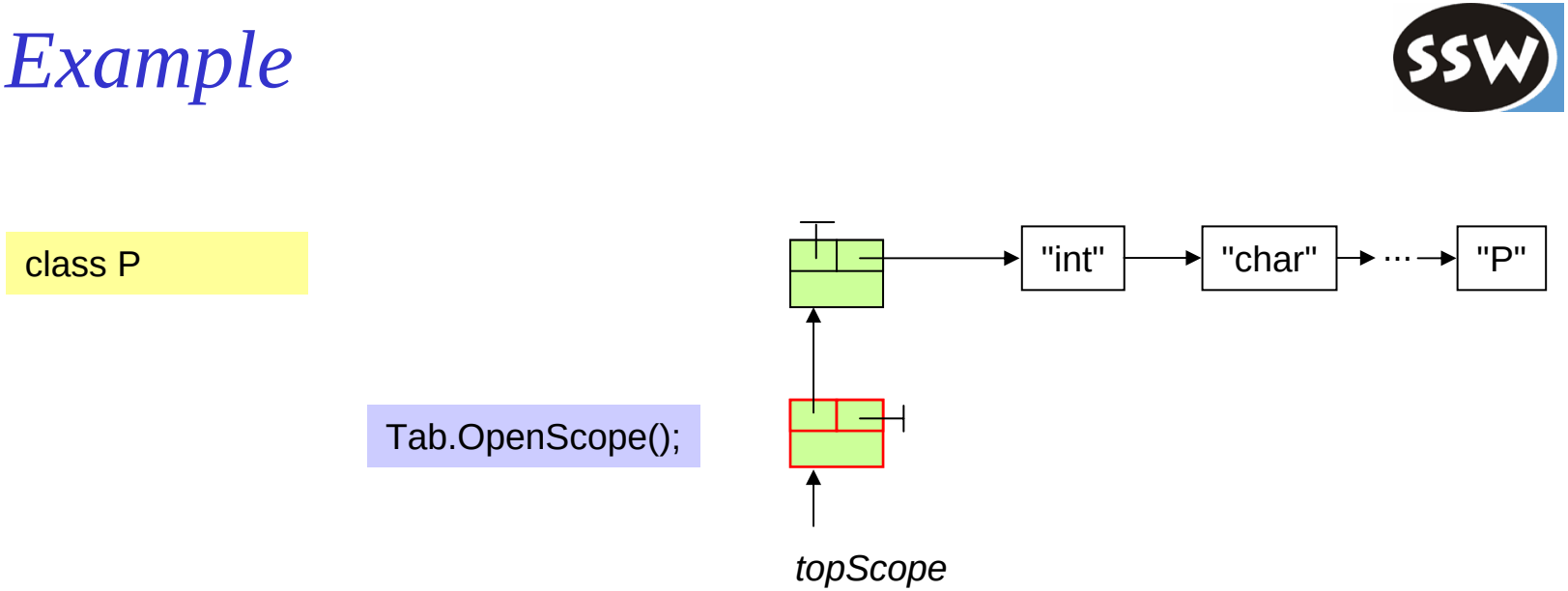

```
MethodDecl                (. Struct type; .)
=  Type<↑type>
   ident                  (.  curMethod = Tab.insert(Symbol.Kinds.Meth, token.str, type);
                              Tab.OpenScope();
                           .)
   ...
   "{"
   ...
   "}"                    (.  curMethod.nArgs = topScope.nArgs;
                              curMethod.nLocs = topScope.nLocs;
                              curMethod.locals = Tab.topScope.locals;
                              Tab.CloseScope();
                           .)
   .
```

global variable

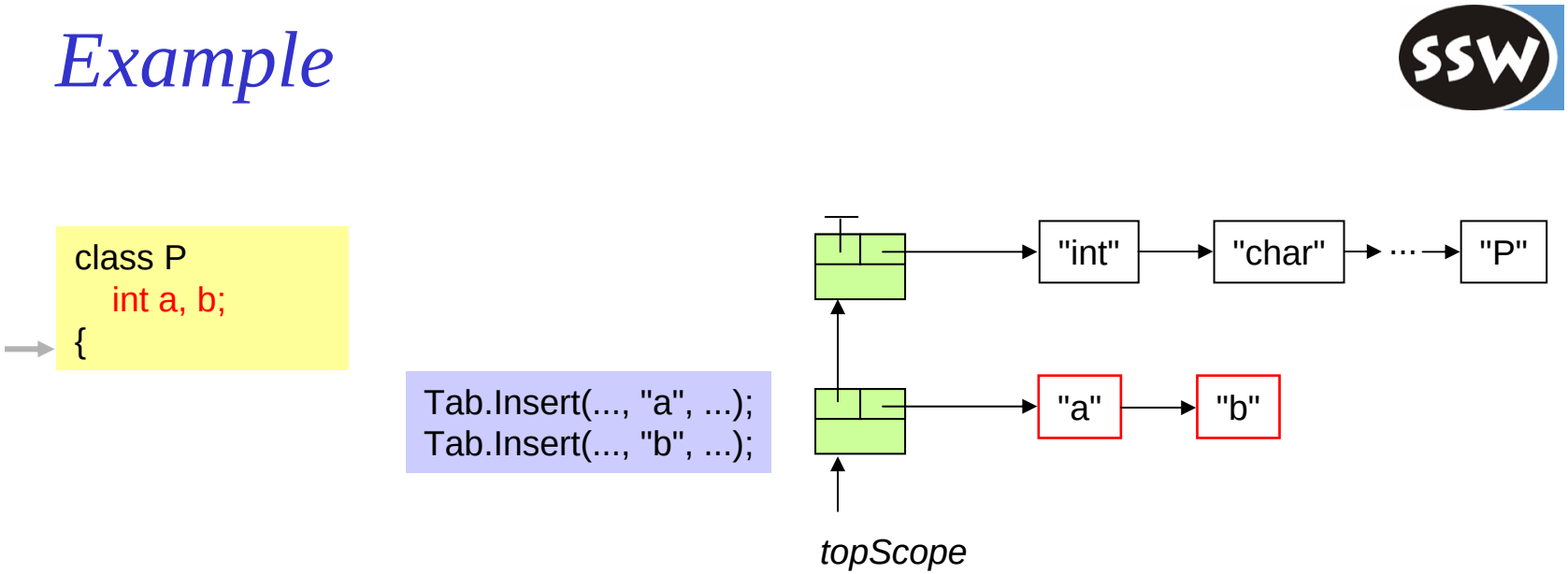
Note

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

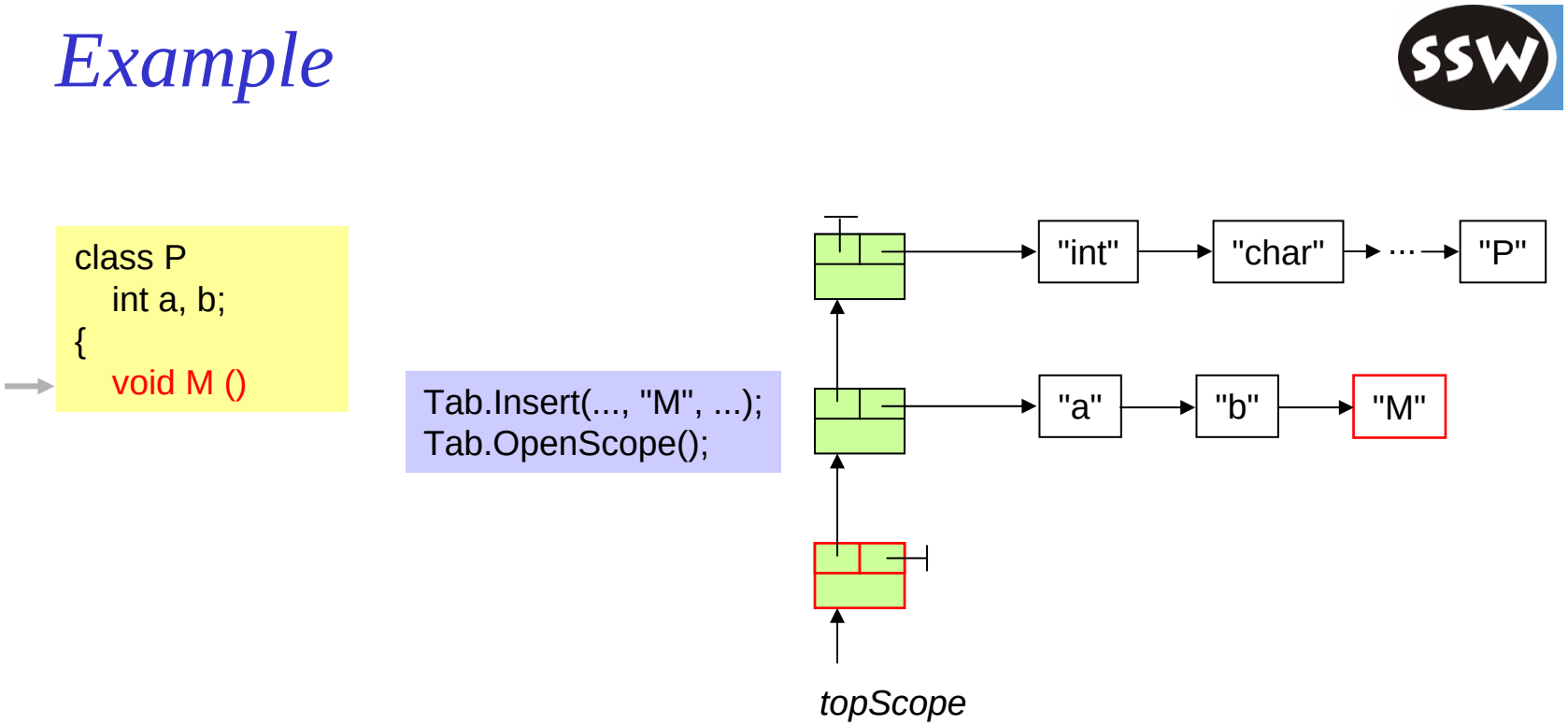
Example



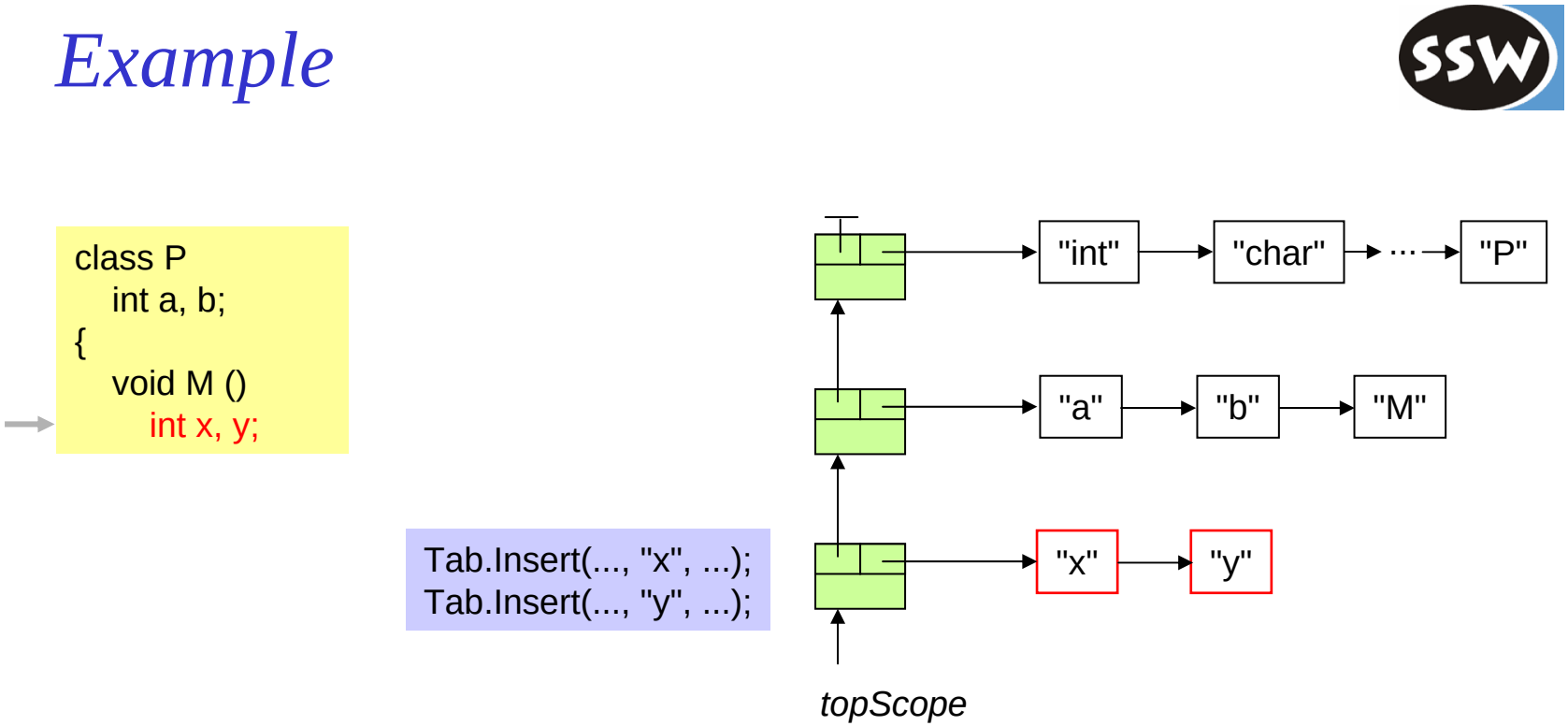
Example



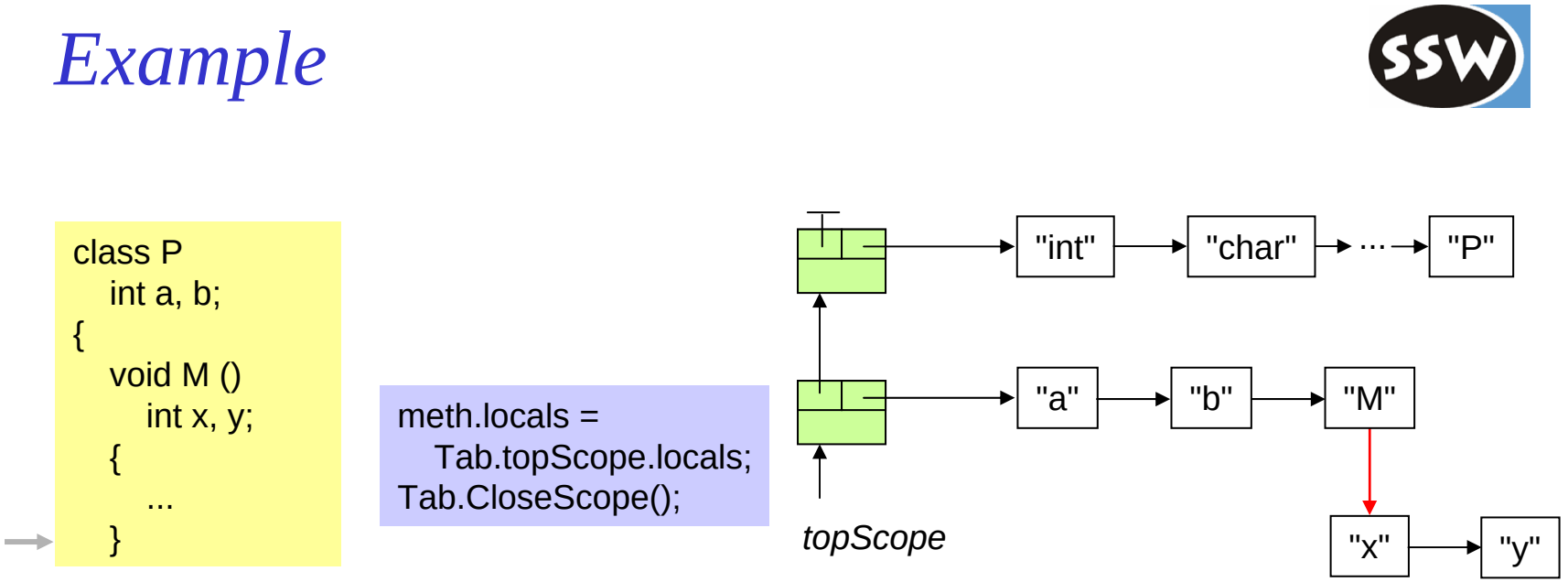
Example


class P
int a, b;
{
void M ()
Tab.Insert(..., "M", ...);
Tab.OpenScope();
"int"
"char"
···
"P"
"a"
"b"
"M"
topScope

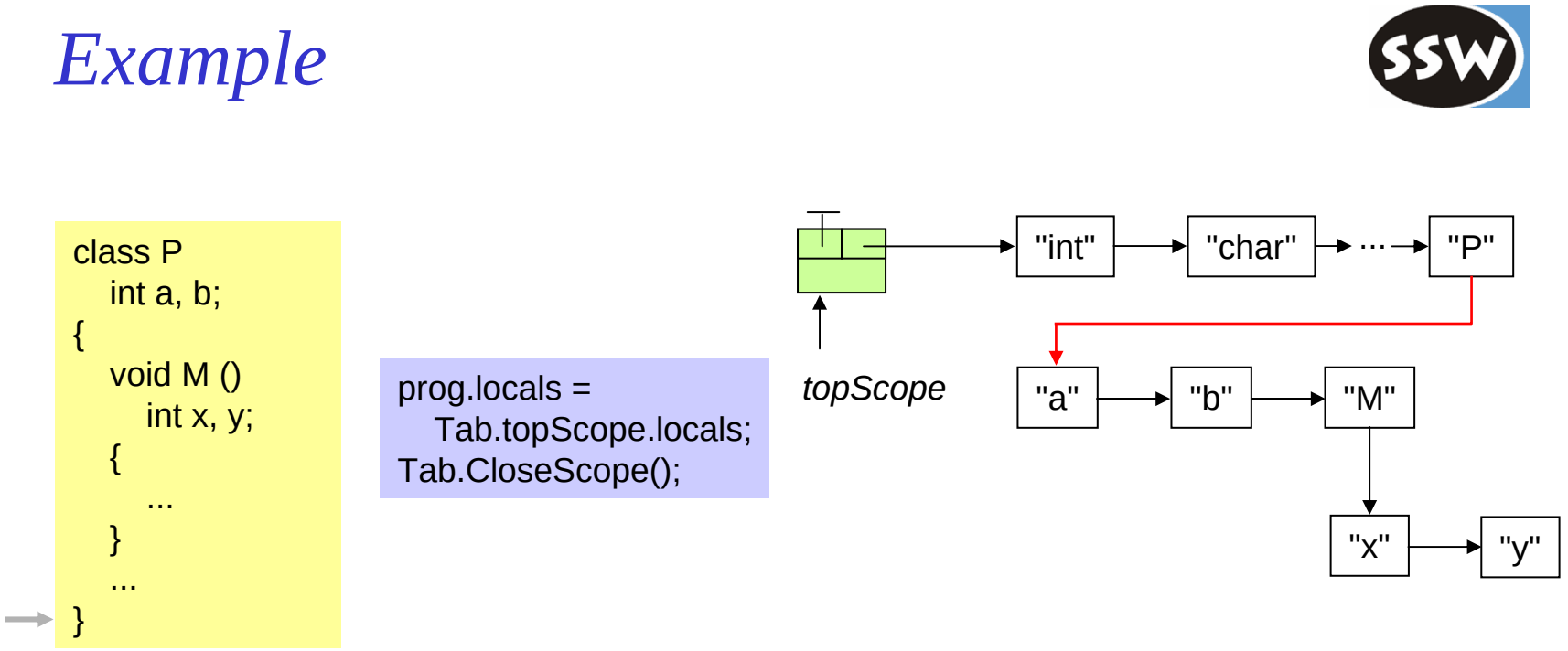
Example


class P
int a, b;
{
void M ()
int x, y;
Tab.Insert(..., "x", ...);
Tab.Insert(..., "y", ...);
"int"
"char"
···
"P"
"a"
"b"
"M"
"x"
"y"
topScope

Example


class P
int a, b;
{
void M ()
int x, y;
{
...
}
meth.locals =
Tab.topScope.locals;
Tab.CloseScope();
"int"
"char"
···
"P"
"a"
"b"
"M"
"x"
"y"
topScope

Example


class P
int a, b;
{
void M ()
int x, y;
{
...
}
...
}
prog.locals =
Tab.topScope.locals;
Tab.CloseScope();
topScope
"int"
"char"
···
"P"
"a"
"b"
"M"
"x"
"y"

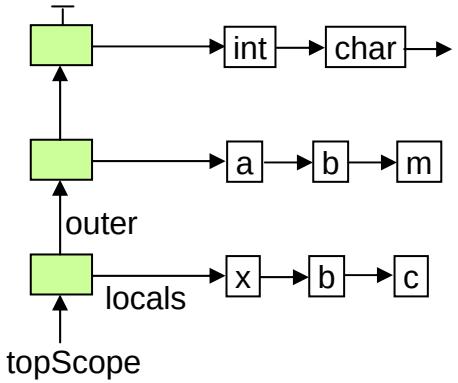
Searching Names in the Symbol Table

The following method is called whenever a name is used

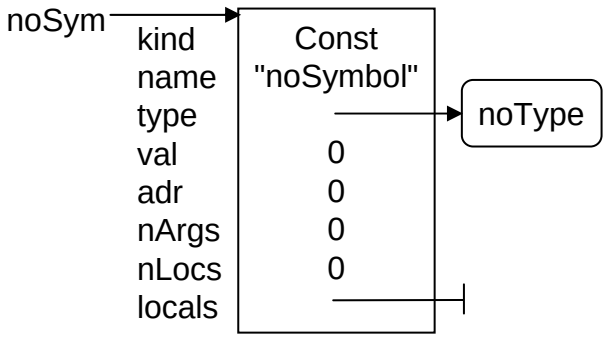
```
Symbol sym = Tab.Find(name);
```

- Lookup starts in *topScope*
- If not found, the lookup is continued in the next outer scope

```
static Symbol Find (string name) {
   for (Scope s = topScope; s != null; s = s.outer)
      for (Symbol sym = s.locals; sym != null; sym = sym.next)
         if (sym.name == name) return sym;
   Parser.Error(name + " is undeclared");
   return noSym;
}
```



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



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

5. Symbol Table

Types

Every object has a type with the following properties

- size (in Z# determined by metadata)
- structure (fields for classes, element type for arrays, ...)

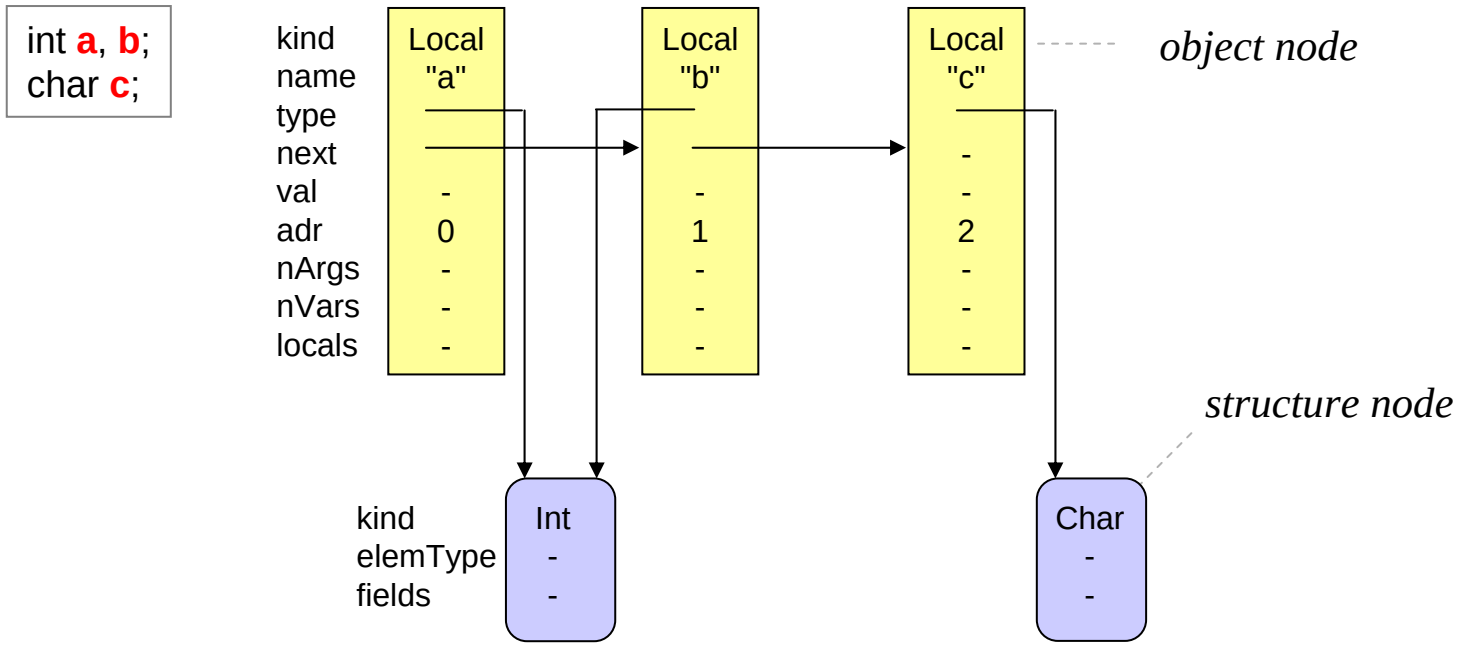
Kinds of types in Z#?

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

Types are represented by structure nodes

```
class Struct {
   public enum Kinds { None, Int, Char, Arr, Class }
   Kinds   kind;
   Struct  elemType;  // Arr: element type
   Symbol fields;     // Class: list of fields
}
```

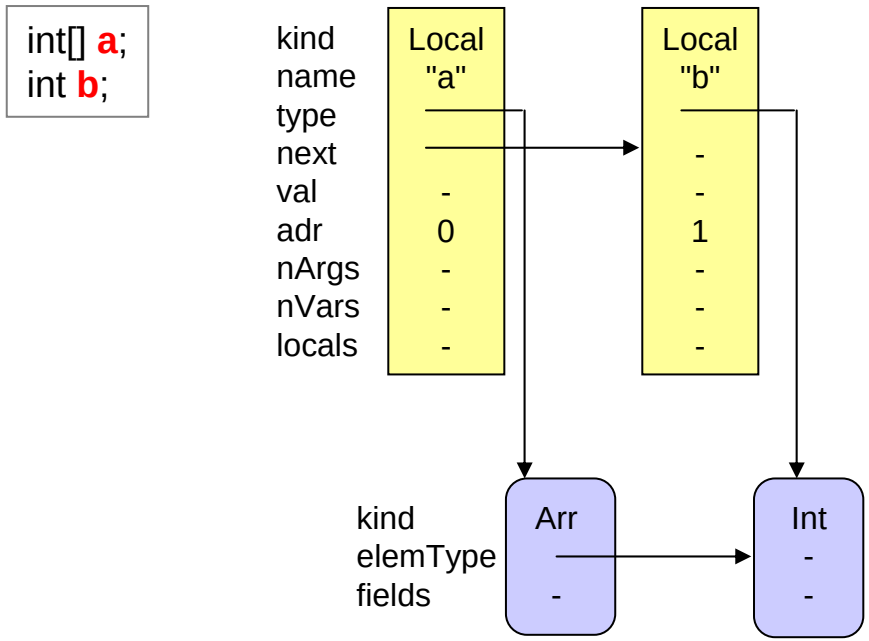
Structure Nodes for Primitive Types



There is just one structure node for *int* in the whole symbol table. All symbols of type *int* reference this one.

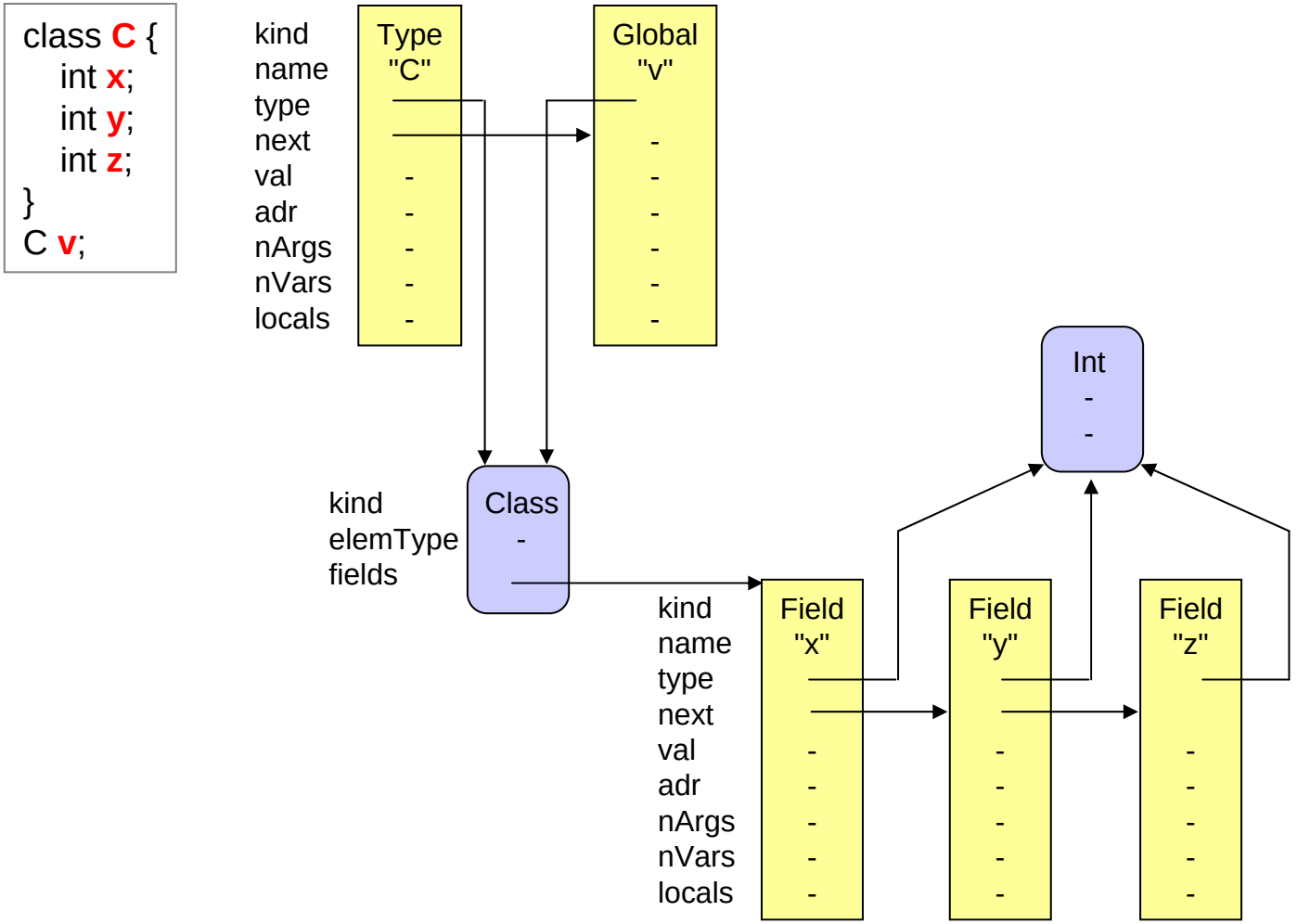
The same is true for structure nodes of type *char*.

Structure Nodes for Arrays



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

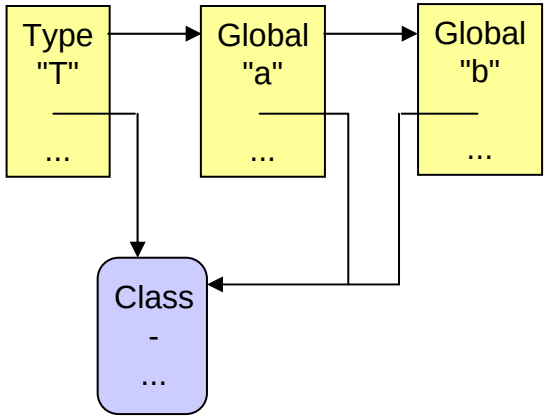
Structure Nodes for Classes



Type Compatibility: *Name Equivalence*

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

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



The types of *a* and *b* are the same

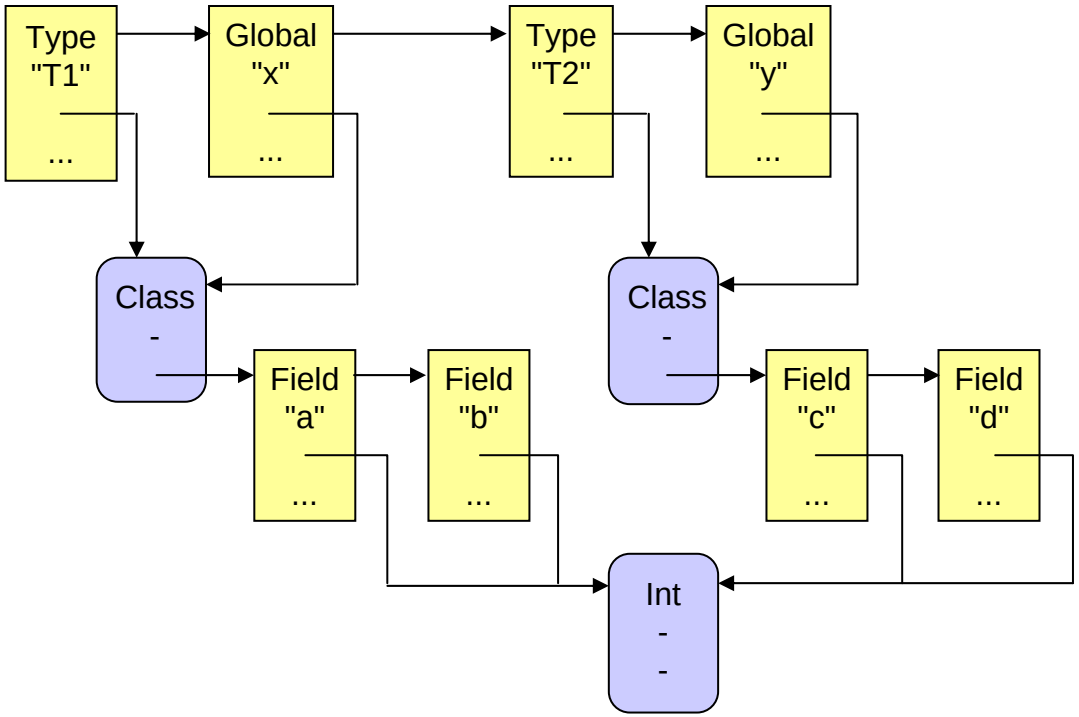
Name equivalence is used in Java, C/C++/C#, Pascal, ..., Z#

Exception
In C# (and Z#) two array types are the same if they have the same element types!

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 equal (but not in Z#!)

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

Methods for Checking Type Compatibility


```
class Struct {
   ...
   // checks, if two types are compatible (e.g. in comparisons)
   public bool CompatibleWith (Struct other) {
      return  this.Equals(other) ||
              this == Tab.nullType && other.IsRefType() ||
              other == Tab.nullType && this.isRefType();
   }

   // checks, if this can be assigned to dest
   public bool AssignableTo (Struct dest) {
      return  this.Equals(dest) ||
              this == Tab.nullType && dest.IsRefType() ||
              kind == Kinds.Arr && dest.kind == Kinds.Arr && dest.elemType == Tab.objType;
   }

   // checks, if two types are equal (structural equivalence for array, name equivalence otherwise)
   public bool Equals (Struct other) {
      if (kind == Kinds.Arr)
         return  other.kind == Kinds.Arr && elemType.Equals(other.elemType);
      return other == this;
   }

   public bool IsRefType() { return kind == Kinds.Class || kind = Kinds.Arr; }
}
```

necessary for standard function len(arr)

Solving LL(1) Conflicts with the Symbol Table

Method syntax in Z#

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

Actually we are used 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

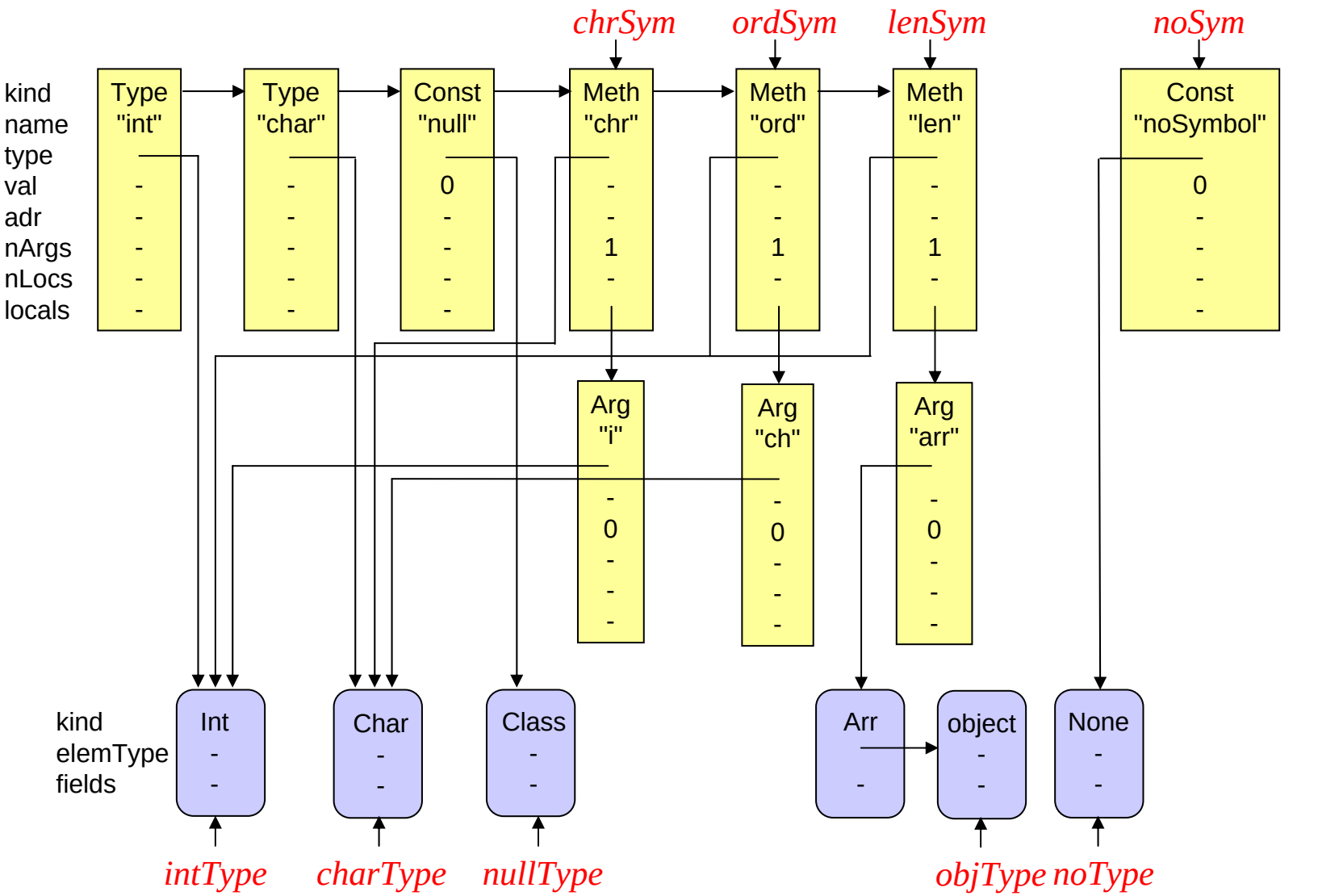

```
static void Block () {
   Check(Token.LBRACE);
   for (;;) {
      if (NextTokenIsType()) VarDecl();
      else if (la ∈ First(Statement)) Statement();
      else if (la ∈ {rbrace, eof}) break;
      else {
         Error("..."); ... recover ...
      }
   }
   Check(Token.RBRACE);
}
```

```
static bool NextTokenIsType() {
   if (la != ident) return false;
   Symbol sym = Tab.Find(laToken.str);
   return sym.kind == Symbol.Kinds.Type;
}
```

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

5. Symbol Table

Structure of the "universe"



Interface of the Symbol Table


```
class Tab {
   static Scope    topScope;     // current top scope

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

   static Symbol   chrSym;       // predefined symbols
   static Symbol   ordSym;
   static Symbol   lenSym;
   static Symbol   noSym;

   static Symbol   Insert (Symbol.Kinds kind, string name, Struct type) {...}
   static Symbol   Find (string name) {...}
   static void     OpenScope () {...}
   static void     CloseScope () {...}

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