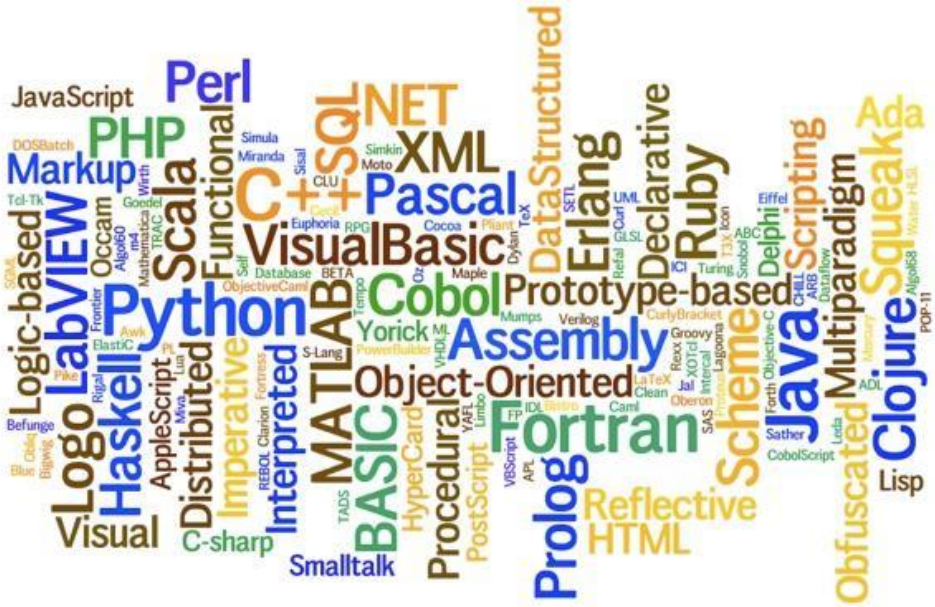


Programming Languages Names, Bindings, and Scopes II

Programming Languages
Module 6

Dr. Tamer ABUHMED
College of Computing

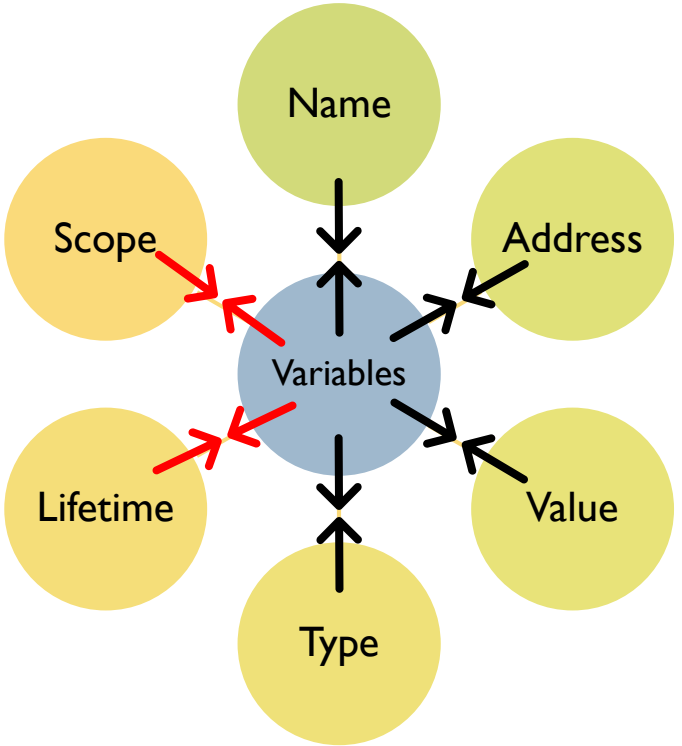
SUNG KYUN KWAN UNIVERSITY

Chapter 5 Topics

- Introduction
- Names
- Variables
- The Concept of Binding
- Scope
- Scope and Lifetime
- Named Constants
- Referencing Environments

Recap: Variables

- Imperative languages are abstractions of von Neumann architecture
  - Memory
  - Processor
  - Variables are characterized by attributes
    - To design a type, must consider scope, lifetime, type checking, initialization, and type compatibility



Scope

- The scope of a declaration or scope of a binding is the region of the program to which a binding applies.
- The *scope* of a variable is the range of statements over which it is visible
- The *local variables* of a program unit are those that are declared in that unit
- The *nonlocal variables* of a program unit are those that are visible in the unit but not declared there
- *Global variables* are a special category of nonlocal variables
- Every language has scope rules which define the scope of declarations, definitions, etc.
- Two broad classes of scope rules:
  - Static or lexical scope - scope determined by structure of program
  - Dynamic scope - scope determined by path of execution

Static or lexical scope

- Based on program code
- To connect a name reference to a variable, you (or the compiler) must find the declaration
- *Search process*: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Enclosing static scopes (to a specific scope) are called its *static ancestors*; the nearest static ancestor is called a *static parent*
- Some languages allow nested subprogram definitions, which create nested static scopes (e.g., Ada, JavaScript, Common LISP, Scheme, Fortran 2003+, F#, and Python)

Nested functions: Python

```
def percent(a, b, c):
    def pc(x):
        return (x*100.0) / (a+b+c)
    print ("Percentages are:", pc(a), pc(b), pc(c))
```

```
def outside(x):
    print(x)
    local = 7
    def inside():
        print("inside",x, local)
    return inside
```

Return function to be evaluated

Static or lexical scope hierarchy

- Global scope
  - The names of all classes defined in the program
- Class scope (OOP languages)
  - Instance scope: all fields and methods of the class
  - Static scope: all static methods
  - Scope of subclass nested in scope of its superclass
- Method scope
  - Formal parameters and local variables in code block of body method
- Code block scope
  - Variables defined in block

Global Scope: Python

- Python
  - A global variable can be referenced in functions, but can be assigned in a function only if it has been declared to be `global` in the function

```
#access global variable in a function
globavar = 5
def f():
    anotherVariable = globvar + 8
```

```
#modifying global variable in a function
globavar = 5
def f():
    global globvar
    globvar +=5
```

Scope Example: C

```
/* what are the scopes of vars? */
int n = 999;
int sub1( ) {
     int n = 10;
     printf("sub1: n = %d\n", n);
     sub2( );
}
int sub2( ) {
     printf("sub2: n = %d\n", n);
}
int main( ) {
     printf("main: n = %d\n", n);
     int n = 50;
     sub1( );
     sub2( );
     printf("main: n = %d\n", n);
}
```

"n" has global scope

"n" has local scope

which "n" has scope here?

"n" has local scope

Scope Example: C++

- In C++, names can be defined in any { ... } block.
- Scope of name is from point of declaration to the end of the block.
- Local scope can create "scope holes" for names in outer scopes. Example: while loop creates scope hole for "**`int x`**".

```
/* what are the scopes of x? */
int sub1( ) {
      int x = 10;
      double sum = 0;
      for(int k=0; k<10; k++) {
          cout << "x = " << x << endl;
          double x = pow(2.0,k);
          sum += x;
      }
      // what is the value of x?
}
```

Scope example

```
class Foo {
  int value;
  int test() {
    int b = 3;
    return value + b;
  }
  void setValue(int c) {
    value = c;
    { int d = c;
      c = c + d;
      value = c;
    }
  }
}

class Bar extends Foo {
  int value;
  void setValue(int c) {
       value = c;
       test();
  }
}
```

scope of local variable `b`

scope of formal parameter `c`

scope of local variable in statement block `d`

scope of field `value`

scope of method `test`

scope of `c`

scope of `value`

Scope Example: Java (1)

- Scope of class members is entire class (can define anywhere)
- Scope of local name is from point of declaration to the end of the block.

```
class A {
    public A(String name) {
        this.name = name;
    }
    public String getName( ) { return name; }
    int sum(int n) {
        int sum = 0; // local name = method name
        for(int k=1; k<=n; k++) {
            sum = sum + k;
        }
        return sum;
    }
    private String name; // defined after use
}
```

Scope Example: Java (2)

- Inside of a method, a block may not redefine a name (flat namespace inside of methods).

```
class A {
    int sum(int n) {
        int sum = 0; // OK
        for(int k=1; k<=n; k++) {
            int sum = 0; // Illegal duplicate
            sum = sum + k++;
        }
        // Error: k is out of scope
        System.out.println( k );
    }
}
```

Scope Example: multiple files

In C and C++ external variables have global scope unless declared "static", which indicates file scope.

File1.c

```
static char *s = "dog";
// external var & funcn
extern int n;
extern int sub2( );

int sub1( ) {
  printf("%s %d", s, n);
  return n;
}
int main( ) {
  int s = 0, n = 0;
  printf("%d", sub2( ));
  printf("%d", sub1( ));
}
```

File2.c

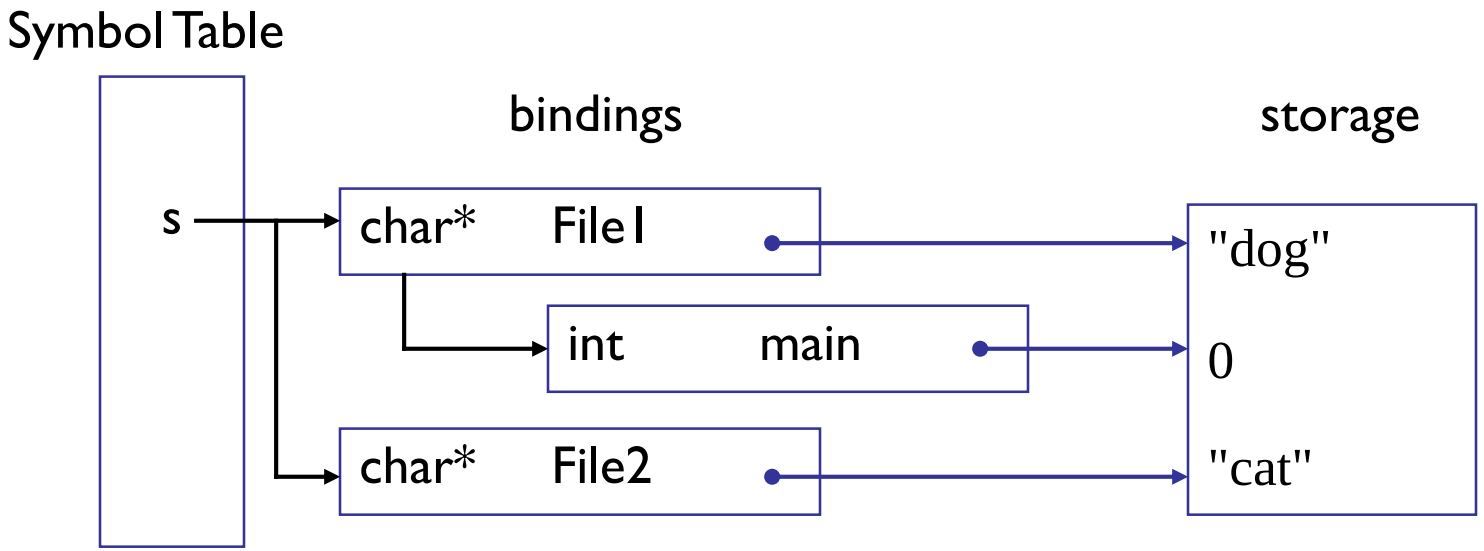
```
static char *s = "cat";
int n = 10;

int sub2( ) {
   printf("%s %d",s, n);
   n = 1000;
   return n;
}
```

What values of s and n will be used in each function?

Scope and Binding

- The binding of names depends on scope rules.
- Previous examples show this clearly.
- The symbol table can include multiple bindings for a variable, depending on scope.



Scope Rules for C, C++, and Java

- C, C++, Java, and most compiled languages use **static scope**.
- Local variables: scope is the block in which variable is declared
  - a *block* is enclosed by { ... }, a function, or a "for( )" loop
- Parameters: scope is the function or method
- Class attributes: scope is the class definition
- Functions (C)
  - global scope
  - must include prototype in other files to use a function
  - linker will resolve function references
- External variables
  - global scope unless declared "`static`"
  - "`static`" externals have file scope
  - "`extern`" *declares* a variable whose *definition* is another file. "`extern`" will not allocate storage for a variable

Scope Example 2

This example contains scope conflicts.
Duplicate names will be detected by the linker (not the compiler).

File1.c

```
char *s = "file1";
int base = 7;

int sub1(int x) {
  printf("sub1 %s",s);
  return x % base;
}
int sub3( ) { return 1;}
int main( ) {
  sub1(10);
  sub2(s);
  sub3( );
}
```

File2.c

```
char *s = "file2";
extern int base;

int sub1(int);
void sub2(char *s) {
    sub1(base+5);
    printf("sub2 %s", s);
}
int sub3( ) {
    printf("sub3 %s", s);
    return 2;
}
```

Dynamic Scope Example (1)

- Perl and some LISP versions use dynamic scope.
- Scope of variable follows path of execution.

```
sub sub1 {
     print "sub1: x = $x\n";   $x = "Elephants";
}
sub sub2 {
     print "sub2: x = $x\n";   $x = "Rats!";
     sub1( );
}
sub main {
     $x = 10;
     print "main: x = $x\n";
     sub1();
     print "main: x = $x\n";
     sub2();
     print "main: x = $x\n";
```

Dynamic Scope Example (2)

- Perl and some LISP versions use dynamic scope.
- Scope of variable follows path of execution.

```
sub sub1 {
     print "sub1: x = $x\n";   $x = "Elephants";
}
sub sub2 {
     print "sub2: x = $x\n";   $x = "Rats!";
     sub1( );
}
sub main {
     $x = 10;
     print "main: x = $x\n";
     sub1();
     print "main: x = $x\n";
     sub2();
     print "main: x = $x\n";
```

```
---------- OUTPUT ----------
main: x = 10
sub1: x = 10
main: x = Elephants
sub2: x = Elephants
sub1: x = Rats!
main: x = Rats!
```

Dynamic Scope Example (3)

- "local" defines a new variable with dynamic scope.

```
sub sub1 {
     print "sub1: x = $x\n";   $x = "Elephants";
}
sub sub2 { local $x;
     print "sub2: x = $x\n";
     $x = "Rats!";
     sub1( );
     print "sub2: x = $x\n";
}
sub main {
     $x = 10;
     print "main: x = $x\n";
     sub2();
     print "main: x = $x\n";
```

```
--------------- OUTPUT ---------------
main: x = 10
sub2: x =
sub1: x = Rats!
sub2: x = Elephants
main: x = 10
```

Dynamic Scope Example (4)

- "my" defines a new variable with *lexical* scope.

```
sub sub1 {
     print "sub1: x = $x\n";   $x = "Elephants";
}
sub sub2 { my $x;
     print "sub2: x = $x\n";
     $x = "Rats!";
     sub1( );
     print "sub2: x = $x\n";
}
sub main {
     $x = 10;
     print "main: x = $x\n";
     sub2();
     print "main: x = $x\n";
```

```
--------------- OUTPUT ---------------
main: x = 10
sub2: x =
sub1: x = 10
sub2: x = Rats!
main: x = Elephants
```

Lexical vs. dynamic scope

- Scope is maintained by the properties of the lookup operation in the symbol table.
- Static (lexical) scope: scope of names is known to the compiler.
  - permits type checking by compiler
  - can easily check for uninitialized variables
  - easier to analyze program correctness
- Dynamic scope: meaning of variables is known only at run-time.
  - cannot perform type checking before execution
  - programs are more flexible, but harder to understand & debug
  - almost always implemented using interpreter (Perl uses a just-in-time compiler, but no type checking)

Scope holes

- A scope hole is a region where a new declaration hides another binding of the same name.

```
class ScopeHole {
    final String s = "global";
    String sub1( ) { return s; }
    String sub2(int which) {
        String s = "local";
        if ( which == 1 ) return sub1();
        else return s;
    }
    void ScopeHole( ) {
        System.out.println("0: s = " + s );
        System.out.println("1: s = " + sub2(1) );
        System.out.println("2: s = " + sub2(2) ); }
```

0: s = global
1: s = global
2: s = local

Why limit scope of names?

- What are advantages of limiting scope of names?

- What would be the problem of making all names be global? Assume all variables are global...

sum.c

```
int n, total;

int sum( ) {
  total = 0;
  for(n=1; n<10; n++)
    total += product(n);

  printf("%d\n",total);
}
```

product.c

```
int n, total;

int product(int k) {
   total = 1;
   for(n=1; n<k; n++)
     total *= n;
   return total;

}
```

Techniques to limit name collision (1)

Limit scope of variables:

- File scope:

```
static int MAX;
```

- Function scope:

```
int sum(int n){
    ...
}
```

- Block scope:

```
for(int k=...){
... }
```

```
/* file scope */
static int MAX = 100;

/* function scope */
int sum( int n ) {
   int total = 0;
   /* block scope in C++
    * but not in standard C
    */
   for(int k=0; k<n; k++) {
      total = total + k;
   }
   return total;
}
```

Techniques to limit name collision (2)

For broader scopes, including scope of functions and variables:

- **Nested** procedures in Pascal, Algol, Ada, ...
- inner procedure can refer to other members of outer procedure.
- scope of inner procedure is limited to the outer procedure.

```
procedure sub1(n: int)
var
   x: real;
   (* nested function *)
   procedure sum(n: int): int
   var
      k, t: int;
   begin
      t := 0;
      for k:=1 to n do
       t := t + k;
      return t;
   end sum;
begin (* start of sub1 *)
   x := sum(20);
   ...
end sub1;
```

Techniques to limit name collision (3)

Modules:

- in Modula 1, 2, 3 and Ada
- module encapsulates both variables and procedures
- a module explicitly "exports" names it wants to make known to outside.
- Usage:

```
var x,y,z: element;
push( x );
push( y );
z := pop( );
```

```
CONST maxsize = 20;
TYPE  element = INT;

MODULE stack;
EXPORT push, pop;
TYPE
   stack_index [1..maxsize];
VAR
   stack: ARRAY stack_index
      of element;
PROCEDURE push(x: element);
begin
   ...
end push;
PROCEDURE pop( ): element;
begin
   ...
end pop;
END stack;
```

Techniques to limit name collision (4)

C++ and C# use "namespace" to encapsulate names.

```
using System;
using System.Data;
using System.Windows.Forms;
namespace MyApplication
{
   public class Form1 {
      public Dimension getSize() {
        ...
      }
      private double width;
      private double height;
      static void Main( ) { ... }
}
```

Techniques to limit name collision (5)

What does Java use to define and use "namespace".

```
/* Java code */
package MyApplication;
import java.util.date;
import  java.awt.*;


public class Form1 {
     public ...
       ...
     }
   }
```

```
/* C# code */
using System;
using System.Data;
using System.Windows.*;
namespace MyApplication
{
   public class Form1 {
      public ...
        ...
      }
   }
```

What is the purpose of "import"?

- What does:

```
import java.util.*;
```

  do?

- Does "import" effect the size of your program?

  For example, would it be more efficient to write:

```
import java.util.Arrays;
```

  instead of

```
import java.util.*;
```

No, "import" statements have no effect on generated byte code and is mainly used for "readability" purposes of your source code

Lifetime

- **Lifetime** is the duration of time that an entity (variable, constant, ...) is bound to memory.
- Lifetime can be...
  - duration of process execution (static variables)
  - duration of object existence (object attributes)
  - duration of function activation (local variables)
  - duration of scope activation (variable defined in a scope). A function also defines a scope.
- Lifetime applies to entities that have values, not names.
- "Lifetime of an identifier" doesn't make sense... identifiers have "scope" rather than "lifetime".

Lifetime and Scope

Scope of identifiers and **lifetime** of entity it refers to are not the same.

- **Lifetime**: time when entity exists in memory
- **scope**: the parts of a program where a name is known or "visible"

```
void add( int count ) {
    static long SUM = 0;
    float x = 0.5;
    while(count>0) {
        int x = 2;
        SUM += x*count;
        count--;
    }
    printf("%f", x);
}
```

`count`: scope is function add( ), lifetime unknown.

`SUM`: scope is function body, lifetime is process execution time.

`float x`: scope is part of the function excluding while loop, lifetime is function activation time.

`int x`: scope is while loop, lifetime is duration of while loop.

Lifetime and Scope (2)

- **`BUFSIZE`** has global scope (any file in this program can refer to its value). Its lifetime is the process's lifetime.
- **`buf`** has file scope. Its lifetime is the program's lifetime.
- **`length`** , **`k`**: scope is the function, lifetime is duration of function.
- **`c`**: scope is the for loop. Life is duration of "for" loop execution.

```
int BUFSIZE = 1024;
static char buf[BUFSIZE];
void read( int length ) {
   int k;
   for(k=0; k<length; k++) {
      int c = getchar(); if (c<0) break;
      buf[k] = c;
   }
   buf[k] = '\0';
}
```

Scope Example: C++

Computer Memory

```
const double PI = 3.14159;
int getPosInt(std::string);
double areaOfCircle(int);
double volOfSphere(int);
int main(){
int radius = getPosInt("Enter a positive integer for the radius of a circle: ");
double aCircle = areaOfCircle(radius);
double vSphere = volOfSphere(radius);
cout << "The area of a circle with r =" << radius << " is: " << aCircle << endl;
cout << "The area of a sphere with r =" << radius << " is: " << vSphere << endl;
return 0;
}
int getPosInt(string msg){
int num = 0;
do{
cout << msg;
cin >> num;
}while(num <= 0);
return num;
} // - calculate the area of a circle based on a radius.   - (Area) A = π r2
double areaOfCircle(int r){ return (PI * pow(r,2));}
```

Computer Memory

```
const double PI = 3.14159;
int getPosInt(std::string);
double areaOfCircle(int);
double volOfSphere(int);
int main(){
int radius = getPosInt("Enter a positive integer for the radius of a circle: ");
double aCircle = areaOfCircle(radius);
double vSphere = volOfSphere(radius);
cout << "The area of a circle with r =" << radius << " is: " << aCircle << endl;
cout << "The area of a sphere with r =" << radius << " is: " << vSphere << endl;
return 0;
}
int getPosInt(string msg){
int num = 0;
do{
cout << msg;
cin >> num;
}while(num <= 0);
return num;
} // - calculate the area of a circle based on a radius.   - (Area) A = π r2
double areaOfCircle(int r){ return (PI * pow(r,2));}
```

PI = 3.14159;

```
const double PI = 3.14159;
int getPosInt(std::string);
double areaOfCircle(int);
double volOfSphere(int);
int main(){
int radius = getPosInt("Enter a positive integer for the radius of a circle: ");
double aCircle = areaOfCircle(radius);
double vSphere = volOfSphere(radius);
cout << "The area of a circle with r =" << radius << " is: " << aCircle << endl;
cout << "The area of a sphere with r =" << radius << " is: " << vSphere << endl;
return 0;
}
int getPosInt(string msg){
int num = 0;
do{
cout << msg;
cin >> num;
}while(num <= 0);
return num;
} // - calculate the area of a circle based on a radius.   - (Area) A = π r2
double areaOfCircle(int r){ return (PI * pow(r,2));}
```

Computer Memory

PI = 3.14159;

main Stack Frame

radius

```
const double PI = 3.14159;
int getPosInt(std::string);
double areaOfCircle(int);
double volOfSphere(int);
int main(){
int radius = getPosInt("Enter a positive integer for the radius of a circle: ");
double aCircle = areaOfCircle(radius);
double vSphere = volOfSphere(radius);
cout << "The area of a circle with r =" << radius << " is: " << aCircle << endl;
cout << "The area of a sphere with r =" << radius << " is: " << vSphere << endl;
return 0;
}
int getPosInt(string msg){
int num = 0;
do{
cout << msg;
cin >> num;
}while(num <= 0);
return num;
} // - calculate the area of a circle based on a radius.   - (Area) A = π r2
double areaOfCircle(int r){ return (PI * pow(r,2));}
```

Computer Memory

PI = 3.14159;

num
msg = "Enter .."

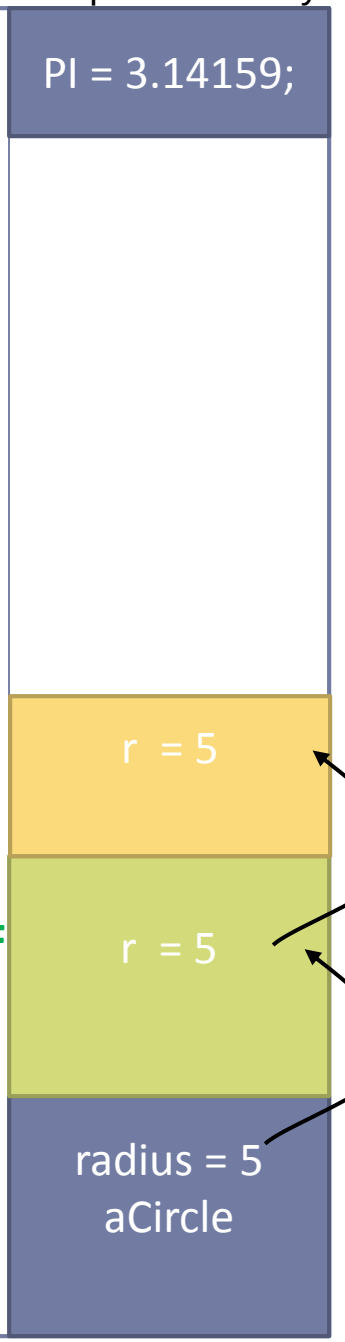
radius

getPosInt S F
Scope??
Life??

main Stack Frame

```
const double PI = 3.14159;
int getPosInt(std::string);
double areaOfCircle(int);
double volOfSphere(int);
int main(){
int radius = getPosInt("Enter a positive integer for the radius of a circle: ");
double aCircle = areaOfCircle(radius);
double vSphere = volOfSphere(radius);
cout << "The area of a circle with r =" << radius << " is: " << aCircle << endl;
cout << "The area of a sphere with r =" << radius << " is: " << vSphere << endl;
return 0;
}
int getPosInt(string msg){
int num = 0;
do{
cout << msg;
cin >> num;
}while(num <= 0);
return num;
} // - calculate the area of a circle based on a radius.   - (Area) A = π r2
double areaOfCircle(int r){ return (PI * pow(r,2));}
```

Computer Memory



pow stack frame

areaOfCircle S F
Scope??
Life??

main S F

```
const double PI = 3.14159;
int getPosInt(std::string);
double areaOfCircle(int);
double volOfSphere(int);
int main(){
int radius = getPosInt("Enter a positive integer for the radius of a circle: ");
double aCircle = areaOfCircle(radius);
double vSphere = volOfSphere(radius);
cout << "The area of a circle with r =" << radius << " is: " << aCircle << endl;
cout << "The area of a sphere with r =" << radius << " is: " << vSphere << endl;
return 0;
}
int getPosInt(string msg){
int num = 0;
do{
cout << msg;
cin >> num;
}while(num <= 0);
return num;
} // - calculate the area of a circle based on a radius.   - (Area) A = π r2
double areaOfCircle(int r){ return (PI * pow(r,2));}
```

Computer Memory

PI = 3.14159;

areaOfCircle S F
Scope??
Life??

r = 5
25

main S F

radius = 5
aCircle

```
const double PI = 3.14159;
int getPosInt(std::string);
double areaOfCircle(int);
double volOfSphere(int);
int main(){
int radius = getPosInt("Enter a positive integer for the radius of a circle: ");
double aCircle = areaOfCircle(radius);
double vSphere = volOfSphere(radius);
cout << "The area of a circle with r =" << radius << " is: " << aCircle << endl;
cout << "The area of a sphere with r =" << radius << " is: " << vSphere << endl;
return 0;
}
int getPosInt(string msg){
int num = 0;
do{
cout << msg;
cin >> num;
}while(num <= 0);
return num;
} // - calculate the area of a circle based on a radius.   - (Area) A = π r2
double areaOfCircle(int r){ return (PI * pow(r,2));}
```

Computer Memory

PI = 3.14159;

r = 5
25
25*PI

areaOfCircle S F
Scope??
Life??

radius = 5
aCircle =78.53

main S F

```
const double PI = 3.14159;
int getPosInt(std::string);
double areaOfCircle(int);
double volOfSphere(int);
int main(){
int radius = getPosInt("Enter a positive integer for the radius of a circle: ");
double aCircle = areaOfCircle(radius);
double vSphere = volOfSphere(radius);
cout << "The area of a circle with r =" << radius << " is: " << aCircle << endl;
cout << "The area of a sphere with r =" << radius << " is: " << vSphere << endl;
return 0;
}
int getPosInt(string msg){
int num = 0;
do{
cout << msg;
cin >> num;
}while(num <= 0);
return num;}
double areaOfCircle(int r){ return (PI * pow(r,2));}
double volOfSphere (int r){ return 4/3 *PI * pow(r,2));}
```

Computer Memory

PI = 3.14159;

main S F

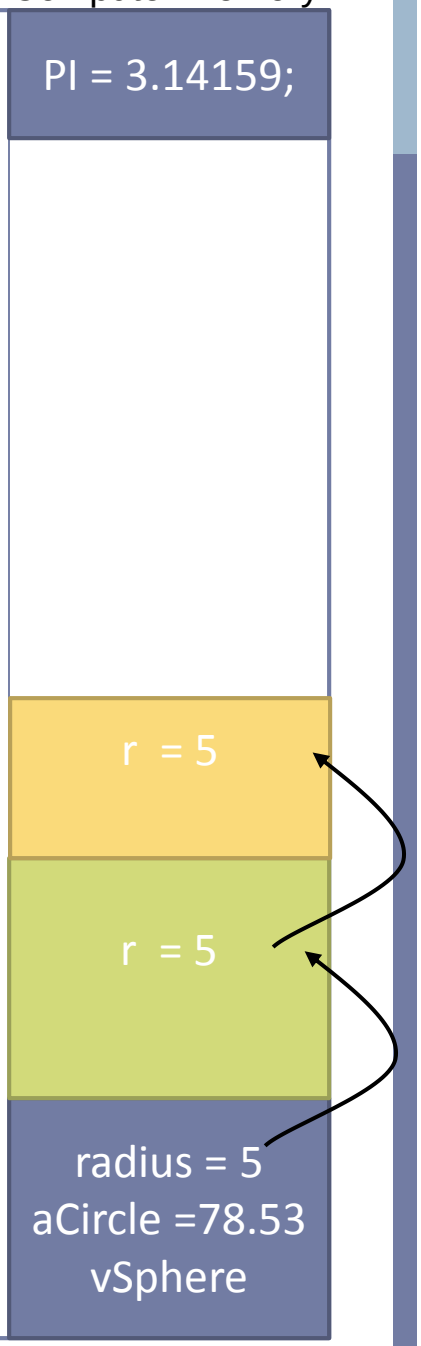
radius = 5
aCircle =78.53
vSphere

```
const double PI = 3.14159;
int getPosInt(std::string);
double areaOfCircle(int);
double volOfSphere(int);
int main(){
int radius = getPosInt("Enter a positive integer for the radius of a circle: ");
double aCircle = areaOfCircle(radius);
double vSphere = volOfSphere(radius);
cout << "The area of a circle with r =" << radius << " is: " << aCircle << endl;
cout << "The area of a sphere with r =" << radius << " is: " << vSphere << endl;
return 0;
}
int getPosInt(string msg){
int num = 0;
do{
cout << msg;
cin >> num;
}while(num <= 0);
return num;}
double areaOfCircle(int r){ return (PI * pow(r,2));}
double volOfSphere (int r){ return 4/3 *PI * pow(r,2));}
```

pow stack frame

volOfSphere S F
Scope??
Life??

main S F



```
const double PI = 3.14159;
int getPosInt(std::string);
double areaOfCircle(int);
double volOfSphere(int);
int main(){
int radius = getPosInt("Enter a positive integer for the radius of a circle: ");
double aCircle = areaOfCircle(radius);
double vSphere = volOfSphere(radius);
cout << "The area of a circle with r =" << radius << " is: " << aCircle << endl;
cout << "The area of a sphere with r =" << radius << " is: " << vSphere << endl;
return 0;
}
int getPosInt(string msg){
int num = 0;
do{
cout << msg;
cin >> num;
}while(num <= 0);
return num;}
double areaOfCircle(int r){ return (PI * pow(r,2));}
double volOfSphere (int r){ return 4/3 *PI * pow(r,2));}
```

Computer Memory

PI = 3.14159;

volOfSphere S F
Scope??
Life??

r = 5
125
4/3*PI*125

main S F

radius = 5
aCircle =78.53
vSphere =

```
const double PI = 3.14159;
int getPosInt(std::string);
double areaOfCircle(int);
double volOfSphere(int);
int main(){
int radius = getPosInt("Enter a positive integer for the radius of a circle: ");
double aCircle = areaOfCircle(radius);
double vSphere = volOfSphere(radius);
cout << "The area of a circle with r =" << radius << " is: " << aCircle << endl;
cout << "The area of a sphere with r =" << radius << " is: " << vSphere << endl;
return 0;
}
int getPosInt(string msg){
int num = 0;
do{
cout << msg;
cin >> num;
}while(num <= 0);
return num;}
double areaOfCircle(int r){ return (PI * pow(r,2));}
double volOfSphere (int r){ return 4/3 *PI * pow(r,2));}
```

Computer Memory

PI = 3.14159;

main Stack Frame

radius = 5
aCircle =78.53
vSphere = 523.598

```
const double PI = 3.14159;
int getPosInt(std::string);
double areaOfCircle(int);
double volOfSphere(int);
int main(){
int radius = getPosInt("Enter a positive integer for the radius of a circle: ");
double aCircle = areaOfCircle(radius);
double vSphere = volOfSphere(radius);
cout << "The area of a circle with r =" << radius << " is: " << aCircle << endl;
cout << "The area of a sphere with r =" << radius << " is: " << vSphere << endl;
return 0;
}
int getPosInt(string msg){
int num = 0;
do{
cout << msg;
cin >> num;
}while(num <= 0);
return num;}
double areaOfCircle(int r){ return (PI * pow(r,2));}
double volOfSphere (int r){ return 4/3 *PI * pow(r,2));}
```

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a `static` variable in a C or C++ function

Constants

- C "const" can be compile time, load time, or run time constants:

```
const int MaxSize = 80;          /* compile time */
void mysub( const int n ) {
     static const NUMBER = 8*sizeof(int);
     const time_t now = time(0);      /* dynamic */
     const int LastN = n;             /* dynamic */
```

- In Java, "final" merely means a variable cannot be changed after the first assignment.

```
final InputStream in = System.in;
void mysub ( int n ) {
     final int LastN = n;
```

Types of Constants

- There are several classes of constants, depending on when their value is known
  - **compile time**: value can be computed at compile time, includes:
    - manifest constants (literals): const int MAX = 1024;
    - computable expressions: const int MAXBYTES = 8*MAX;
  - **elaboration time**: value is determined when the program is executed.
    - sometimes this simply means a variable whose value cannot be changed (this is enforced by the compiler).

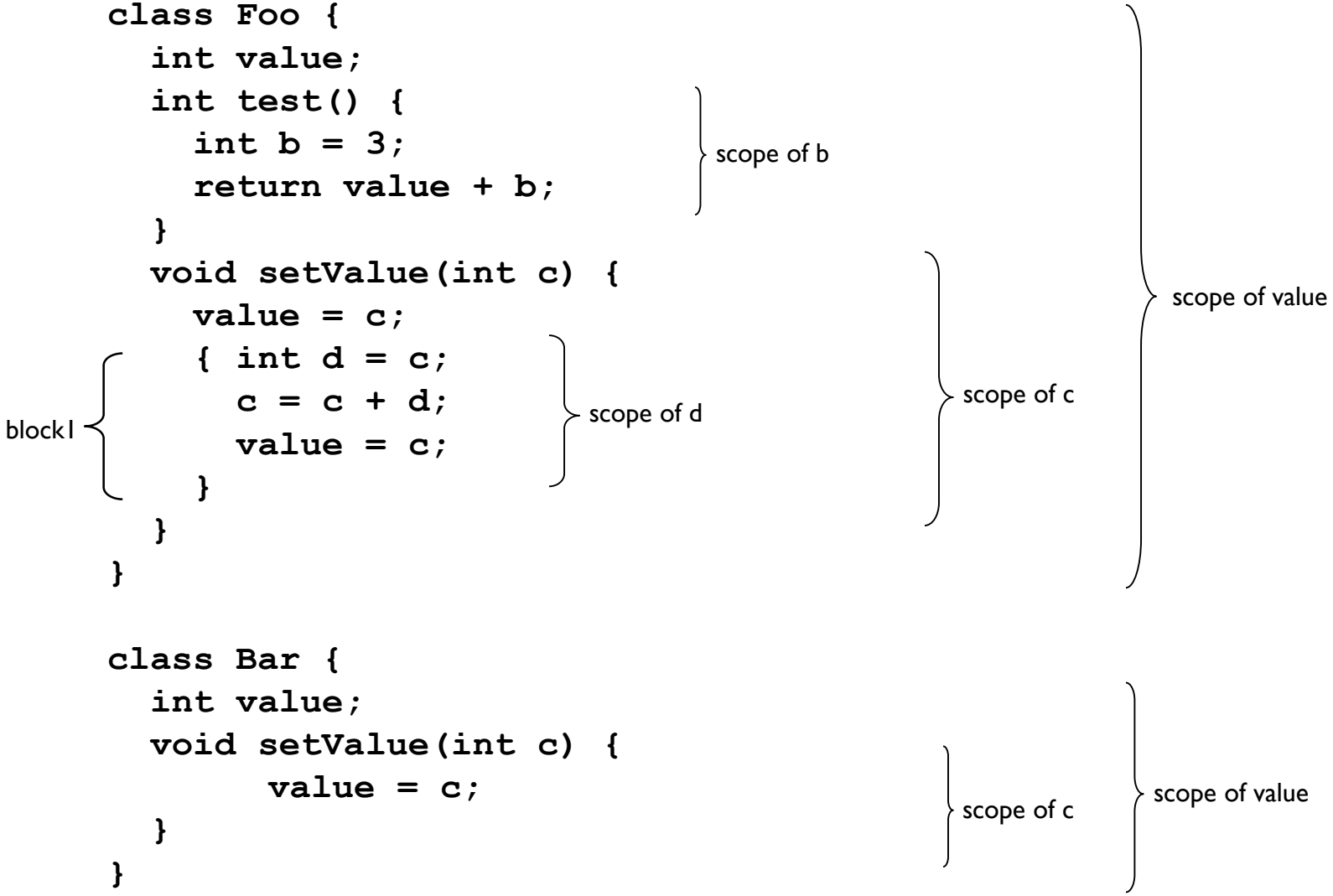
Declaration Order

- C99, C++, Java, and C# allow variable declarations to appear anywhere a statement can appear
  - In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block
  - In C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block
    - However, a variable still must be declared before it can be used

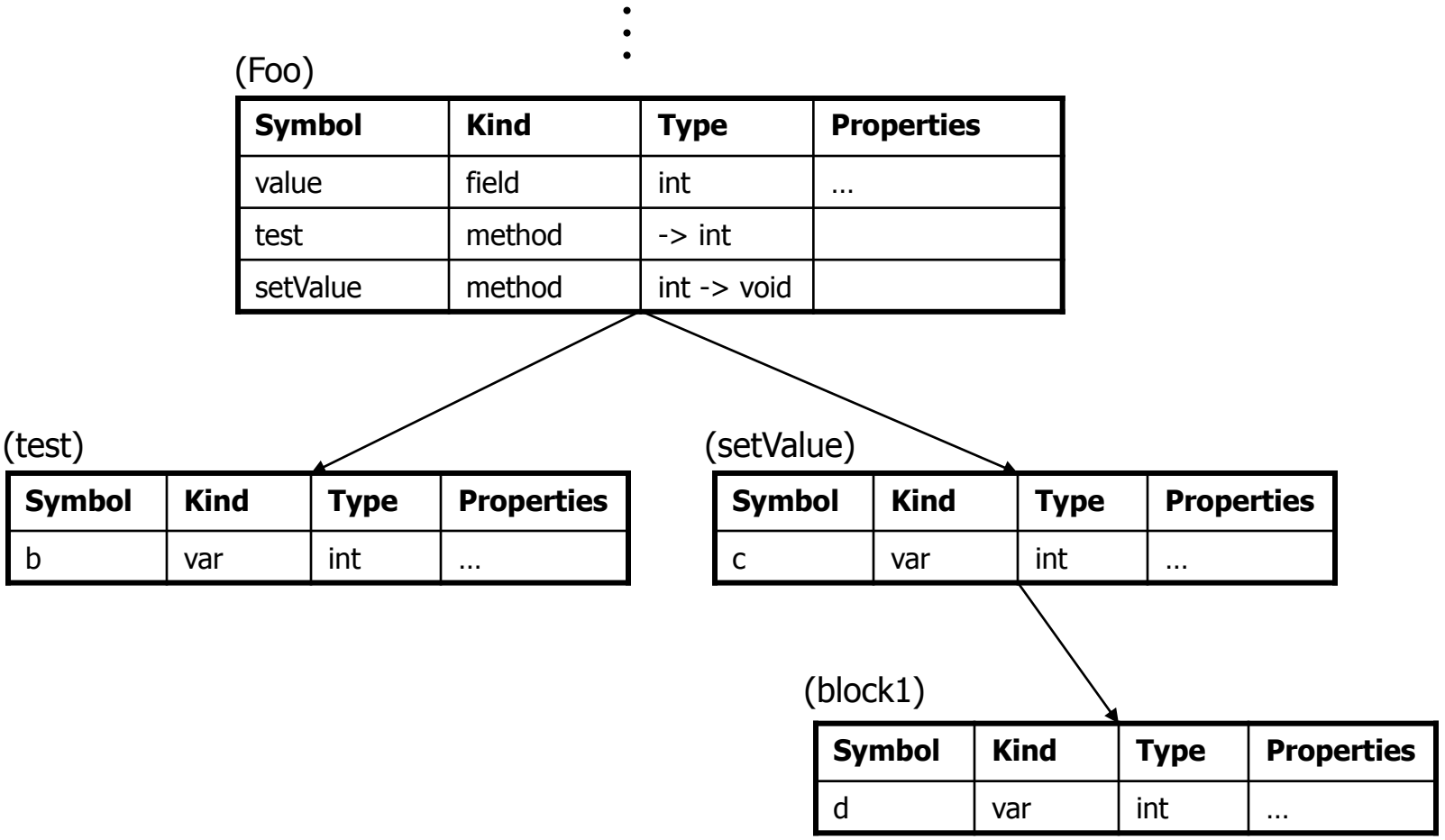
Referencing Environments

- The *referencing environment* of a statement is the collection of all names that are visible in the statement
- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- A subprogram is active if its execution has begun but has not yet terminated
- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms

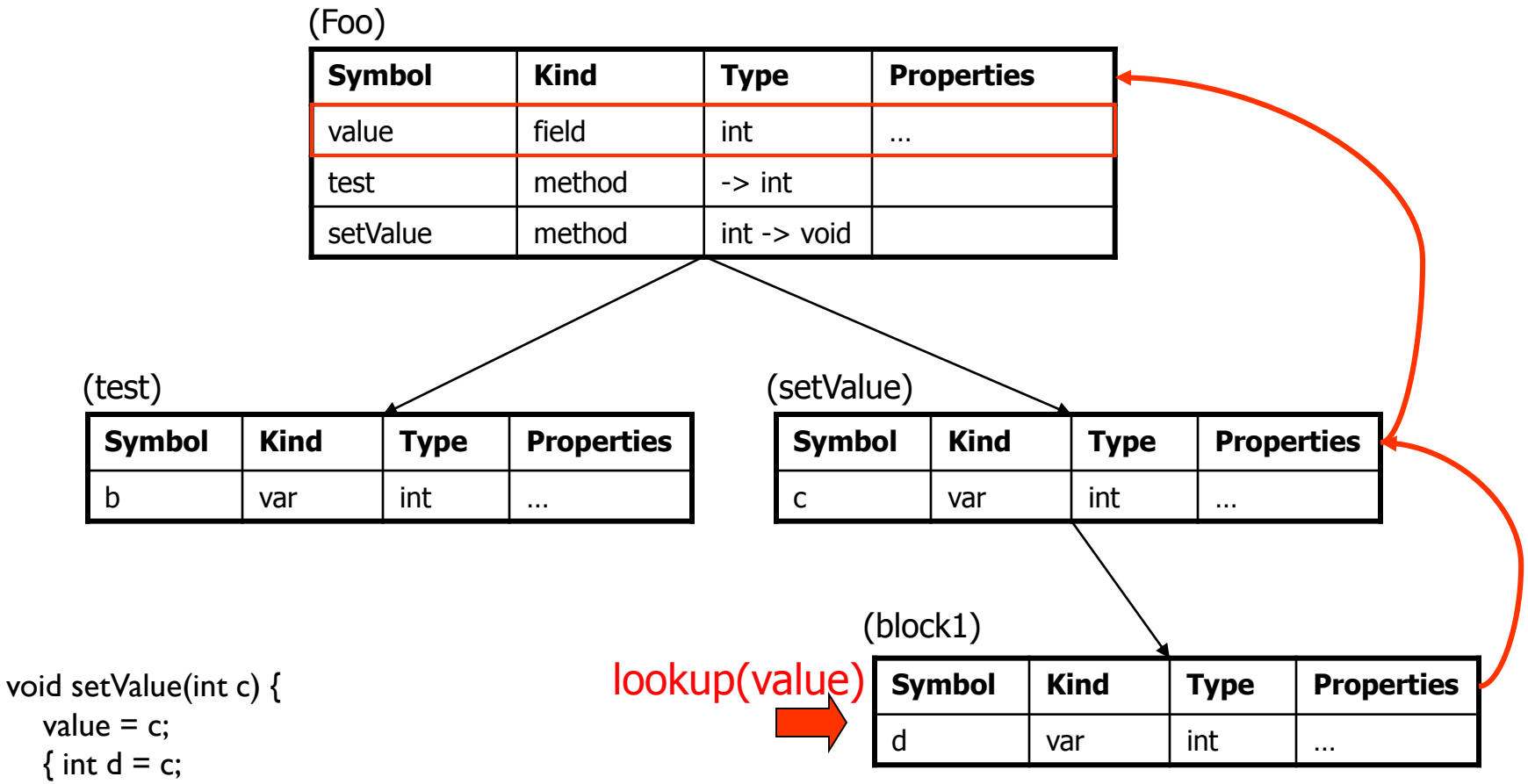
Symbol table example



Symbol table example cont.



Checking scope rules




```
void setValue(int c) {
    value = c;
    { int d = c;
      c = c + d;
      value = c;
    }
}
```

Chapter 5 Summary

- Introduction
- Names
- Variables
- The Concept of Binding
- Scope
- Scope and Lifetime
- Named Constants
- Referencing Environments