

Semantics

CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

Program Errors

- Program is lexically well-formed
 - Identifiers have valid names
 - Strings are properly terminated
 - No unknown characters
- Program is syntactically well-formed:
 - Package declaration have the correct structure
 - Expressions are syntactically valid
- Does this mean that the program is legal?

Example (decaf program)

```
package test {  
    var myBin bool;  
    func foo() void {  
        var x[0] int;  
        var k int = myBin * y;  
    }  
    func foo() void {  
    }  
    func fibonacci(n int) int {  
        return foo() + fibonacci(n-1);  
    }  
}
```

Example (decaf program)

```
package test {
```

```
  var myBin bool;
```

```
  func foo() void {
```

```
    var x[0] int;
```

```
    var k int = myBin * y;
```

```
  }
```

```
  func foo() void {
```

```
  }
```

```
  func fibonacci(n int) int {
```

```
    return foo() + fibonacci(n-1);
```

```
  }
```

```
  [redacted]
```

```
}
```

Cannot define Array type as local variable
Cannot define Array of size 0

Variable not declared

Cannot redefine
functions

Cannot multiply
boolean

Cannot add void

No main function

Goal of Semantic Analysis

- Ensure that the program has a well-defined meaning
- Verifies properties of the program that are not caught during the earlier phases
 - All variables are declared before use
 - Types are used correctly in expressions
 - Method calls have correct number and types of parameters and return value


Challenges in Semantic Analysis

- Reject all/most of the incorrect programs
- Accept all correct programs

Validity vs Correctness

```
func main () int {  
    var x string;  
    if (false) {  
        x = 137;  
    }  
}
```

Safe! cannot
happen!



Validity vs Correctness

```
func fibonacci (n int) int {  
    if ( n<=1 ) return 0;  
    return fibonacci (n-1) + fibonacci(n-2);  
}  
  
func main() int {  
    print_int (fibonacci(40));  
}
```

Incorrect! Should be "return n;"

Challenges in Semantic Analysis

- Reject the largest number of incorrect programs
- Accept all correct programs
- Work fast!

Other Goals of Semantic Analysis

- Gather useful information about the program for code generation:
 - Determine what variables are meant by each identifier
 - Build an internal representation of inheritance hierarchies
 - Keep track of variables which are in scope at each program point

Limitation of CFGs

- Using CFGs
 - How would you prevent duplicate package definitions?
 - How would you differentiate variables of one type from variables of another type?
 - How would you ensure all called methods are defined?
- For most programming languages, these are *provably impossible* in a CFG

Implementing Semantic Analysis

- Attribute Grammars
 - Augment parsing rules to do checking during parsing
 - Has its limitations
- Recursive AST Walk
 - Construct the AST, then use recursion to explore the tree

Scoping

What's in a Name?

- The same name (identifier) in a program may refer to fundamentally different things:
- This is perfectly legal Java code:

```
public class A {  
    char A;  
    A A (A A) {  
        A.A = 'A';  
        return A ( (A) A);  
    }  
}
```

What's in a Name?

- The same name (identifier) in a program may refer to fundamentally different things:
- This is perfectly legal Java code:

```
public class A {  
    char A;  
    A A (A A) {  
        A.A = 'A';  
        return A ( (A) A);  
    }  
}
```

What's in a Name?

- The same name (identifier) in a program may refer to completely different objects:
- This is perfectly legal C++ code:

```
int Awful () {  
    int x = 137;  
    {  
        string x = "Scope!"  
        if (float x = 0)  
            double x = x;  
    }  
    if (x == 137) cout << "Y";  
}
```


What's in a Name?

- The same name (identifier) in a program may refer to completely different objects:
- This is perfectly legal C++ code:

```
int Awful () {  
    int x = 137;  
    {  
        string x = "Scope!"  
        if (float x = 0)  
            double x = x;  
    }  
    if (x == 137) cout << "Y";  
}
```

Scope

- The **scope** of an entity is the set of locations in a program where that entity's name refers to that entity.
- The introduction of new variables into scope may hide older variables
- How do we keep track of what's visible?

Symbol Tables

- Symbol tables map **names** (string format) to **descriptors** (information about identifiers)
- As we run our semantic analysis, continuously update the symbol table with information about what is in scope

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y){
3:     printf(“%d, %d, %d\n”, x, y, z);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:            {
11:                printf(“%d, %d, %d\n”, x, y, z);
12:            }
13:            printf(“%d, %d, %d\n”, x, y, z);
14:        }
15:        printf(“%d, %d, %d\n”, x, y, z);
16:    }
17: }
```

Symbol Tables

Symbol Table

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y){
3:     printf("%d, %d, %d\n", x, y, z);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y){
3:     printf("%d, %d, %d\n", x, y, z);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y){
3:     printf("%d, %d, %d\n", x, y, z);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x, y, z);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x, y, z);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x, y, z);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x, y, z);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x = z;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x = z;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5
Y	9

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:        {
11:            printf("%d, %d, %d\n", x, y, z);
12:        }
13:        printf("%d, %d, %d\n", x, y, z);
14:    }
15:    printf("%d, %d, %d\n", x, y, z);
16: }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5
Y	9

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5
Y	9

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:            {
11:                printf("%d, %d, %d\n", x, y, z);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5
Y	9

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:            {
11:                printf("%d, %d, %d\n", x @5, y @9, z @5);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5
Y	9

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:            {
11:                printf("%d, %d, %d\n", x @5, y @9, z @5);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5
Y	9

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:            {
11:                printf("%d, %d, %d\n", x @5, y @9, z @5);
12:            }
13:            printf("%d, %d, %d\n", x, y, z);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5
Y	9

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:            {
11:                printf("%d, %d, %d\n", x @5, y @9, z @5);
12:            }
13:            printf("%d, %d, %d\n", x @5, y @9, z @5);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5
Y	9

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:            {
11:                printf("%d, %d, %d\n", x @5, y @9, z @5);
12:            }
13:            printf("%d, %d, %d\n", x @5, y @9, z @5);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:            {
11:                printf("%d, %d, %d\n", x @5, y @9, z @5);
12:            }
13:            printf("%d, %d, %d\n", x @5, y @9, z @5);
14:        }
15:        printf("%d, %d, %d\n", x, y, z);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:            {
11:                printf("%d, %d, %d\n", x @5, y @9, z @5);
12:            }
13:            printf("%d, %d, %d\n", x @5, y @9, z @5);
14:        }
15:        printf("%d, %d, %d\n", x @5, y @2, z @5);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:            {
11:                printf("%d, %d, %d\n", x @5, y @9, z @5);
12:            }
13:            printf("%d, %d, %d\n", x @5, y @9, z @5);
14:        }
15:        printf("%d, %d, %d\n", x @5, y @2, z @5);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2
X	5
Z	5

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:            {
11:                printf("%d, %d, %d\n", x @5, y @9, z @5);
12:            }
13:            printf("%d, %d, %d\n", x @5, y @9, z @5);
14:        }
15:        printf("%d, %d, %d\n", x @5, y @2, z @5);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:            {
11:                printf("%d, %d, %d\n", x @5, y @9, z @5);
12:            }
13:            printf("%d, %d, %d\n", x @5, y @9, z @5);
14:        }
15:        printf("%d, %d, %d\n", x @5, y @2, z @5);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:            {
11:                printf("%d, %d, %d\n", x @5, y @9, z @5);
12:            }
13:            printf("%d, %d, %d\n", x @5, y @9, z @5);
14:        }
15:        printf("%d, %d, %d\n", x @5, y @2, z @5);
16:    }
17: }
```

Symbol Table	
X	0
Z	1
X	2
Y	2

Symbol Tables

```
0: int x = 137;
1: int z = 42;
2: int testFunc(int x, int y) {
3:     printf("%d, %d, %d\n", x @2, y @2, z @1);
4:     {
5:         int x, z;
6:         z @5 = y @2;
7:         x @5 = z @5;
8:         {
9:             int y = x @5;
10:            {
11:                printf("%d, %d, %d\n", x @5, y @9, z @5);
12:            }
13:            printf("%d, %d, %d\n", x @5, y @9, z @5);
14:        }
15:        printf("%d, %d, %d\n", x @5, y @2, z @5);
16:    }
17: }
```

Symbol Table	
X	0
Z	1

Symbol Tables

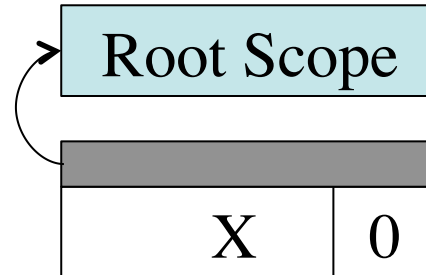
- Symbol tables map **names** (string format) to **descriptors** (information about identifiers)
- As we run our semantic analysis, continuously update the symbol table with information about what is in scope
- Typical implementation: stack
- Basic Operations:
 - Push scope: Enter a new scope
 - Pop scope: Leave a scope, discarding all declarations
 - Insert symbol: add a new identifier to the current scope
 - Lookup symbol: Given an identifier, find a descriptor

Using a Symbol Table

- To process a portion of the program that creates a scope (block statements, function calls, classes, etc.)
 - Enter a new scope
 - Add all variable declarations to the symbol table
 - Process the body of the block/function/class
 - Exit the scope
- Much of semantic analysis is defined over the parse tree using symbol tables

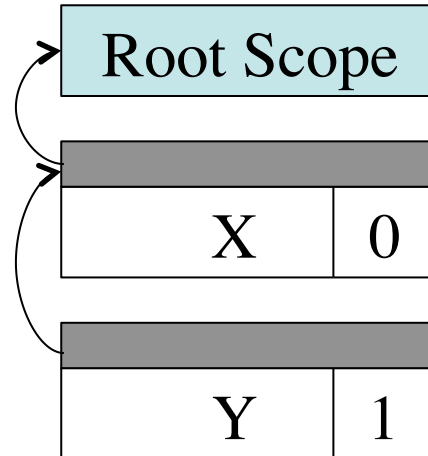
Another View of Symbol Table

→ 0: int x;
1: int y;
2: int testFunc(int x, int y)
3: {
4: int w, z;
5: {
6: int y;
7: }
8: {
9: int w;
10: }
11: }



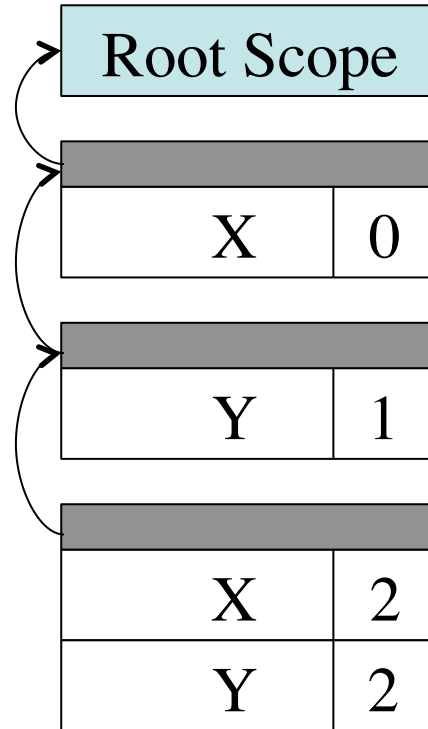
Another View of Symbol Table

0: int x;
1: int y;
2: int testFunc(int x, int y)
3: {
4: int w, z;
5: {
6: int y;
7: }
8: {
9: int w;
10: }
11: }



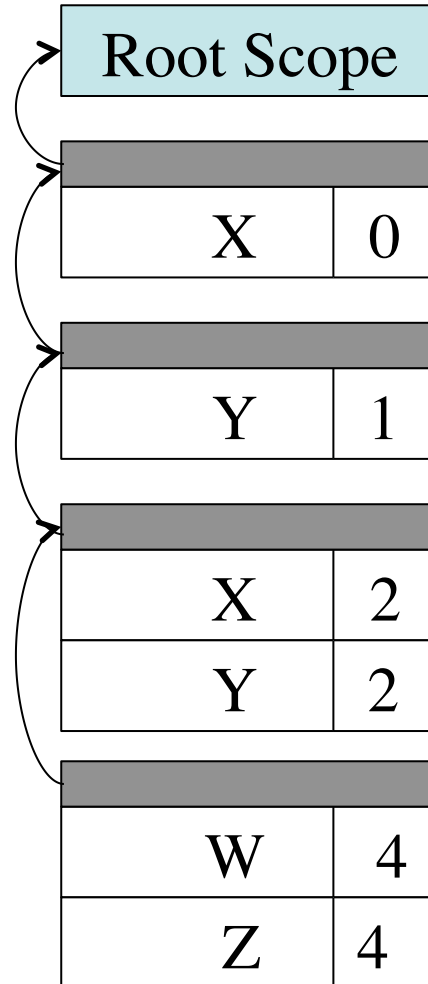
Another View of Symbol Table

```
0: int x;  
1: int y;  
➔ 2: int testFunc(int x, int y)  
3: {  
4:     int w, z;  
5:     {  
6:         int y;  
7:     }  
8:     {  
9:         int w;  
10:    }  
11: }
```

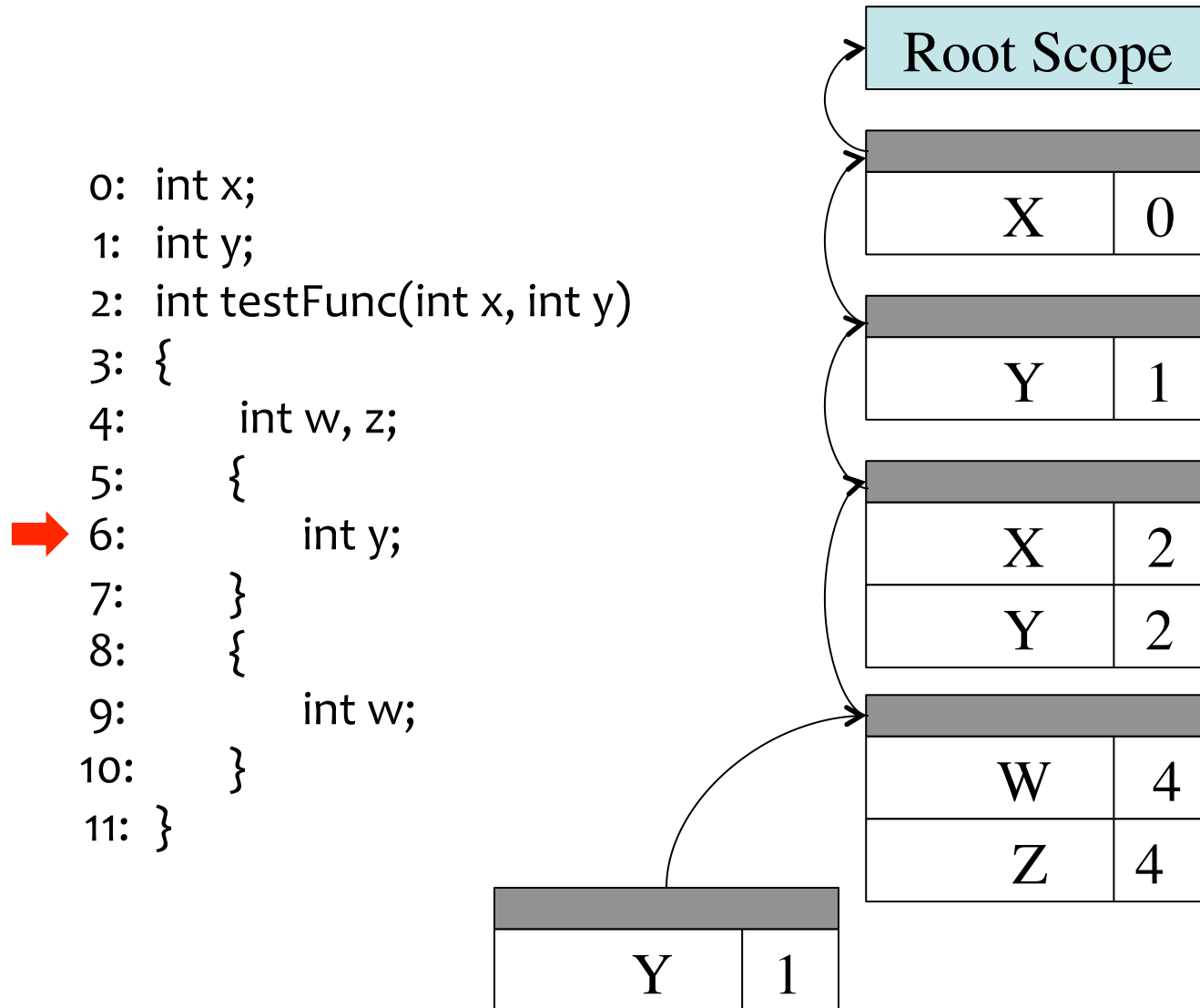


Another View of Symbol Table

```
0: int x;  
1: int y;  
2: int testFunc(int x, int y)  
3: {  
4:     int w, z;  
5:     {  
6:         int y;  
7:     }  
8:     {  
9:         int w;  
10:    }  
11: }
```

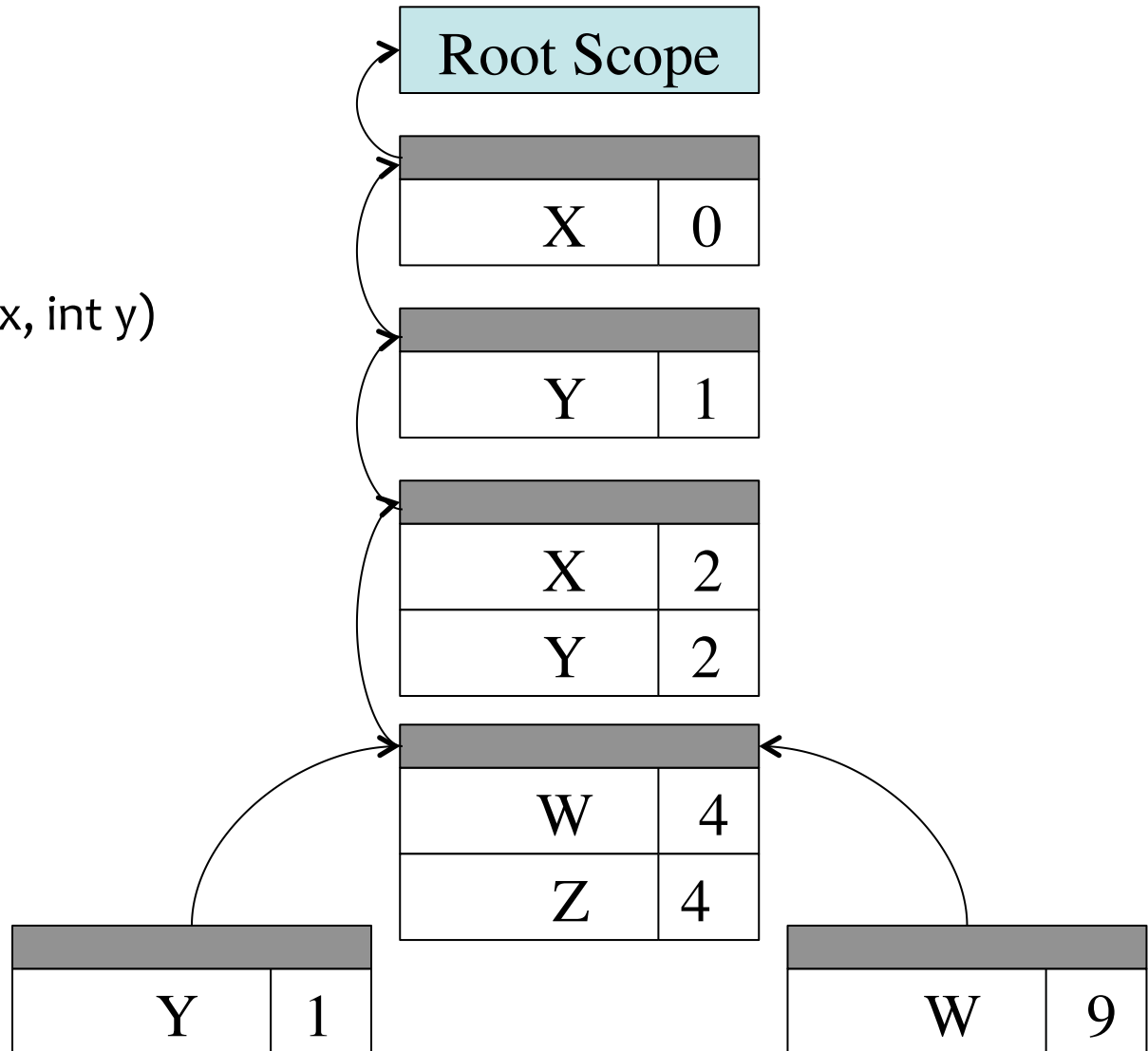


Another View of Symbol Table



Another View of Symbol Table

```
0: int x;  
1: int y;  
2: int testFunc(int x, int y)  
3: {  
4:     int w, z;  
5:     {  
6:         int y;  
7:     }  
8:     {  
9:         int w;  
10:    }  
11: }
```



Spaghetti Stacks

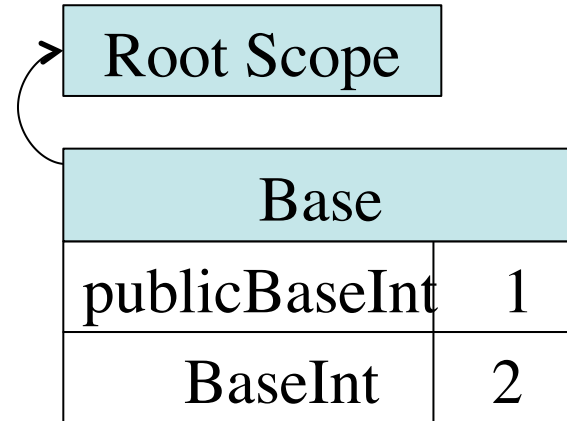
- Treat the symbol table as a linked structure of scopes
- Each scope stores a pointer to its parent, but not vice-versa
- From any point in the program, symbol table appears to be a stack
- This is called a spaghetti stack

Why Two Interpretations?

- Spaghetti stack is a *static structure*; explicit stack is *dynamic structure*.
- Spaghetti stack can be stored in the abstract syntax tree data structure for a program.
- Explicit stack uses less memory and is better for recursive function invocations.

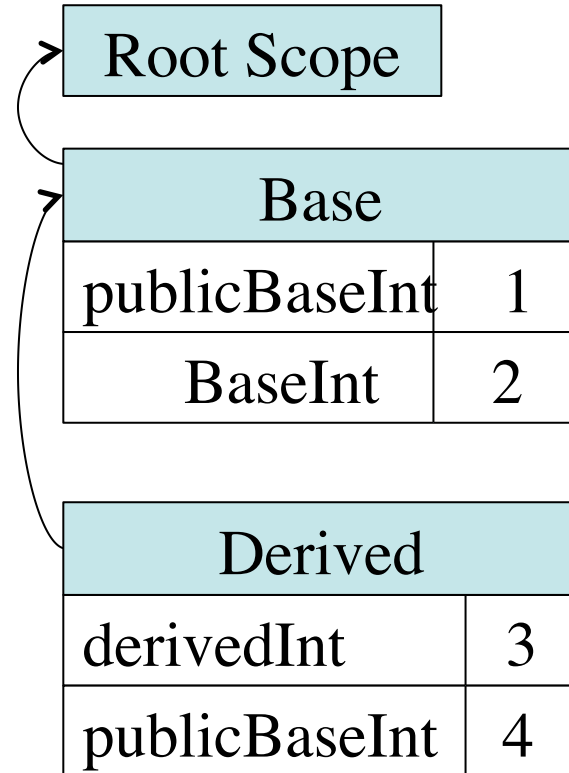
Scoping with Inheritance

```
public class Base {  
    public int publicBaseInt = 1;  
    protected int baseInt = 2;  
}
```



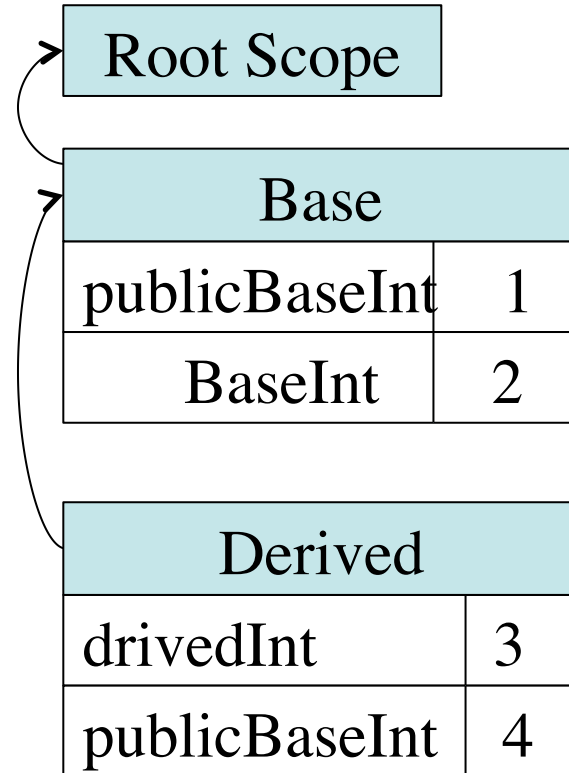
Scoping with Inheritance

```
public class Base {  
    public int publicBaseInt = 1;  
    protected int baseInt = 2;  
}  
  
public class Derived extends Base {  
    public int derivedInt = 3;  
    public int publicBaseInt = 4;  
  
    public void doSomething () {  
        System.out.println(publicBaseInt);  
        System.out.println(baseInt);  
        System.out.println(derivedInt);  
  
        int publicBaseInt = 6;  
        System.out.println(publicBaseInt);  
    }  
}
```



Scoping with Inheritance

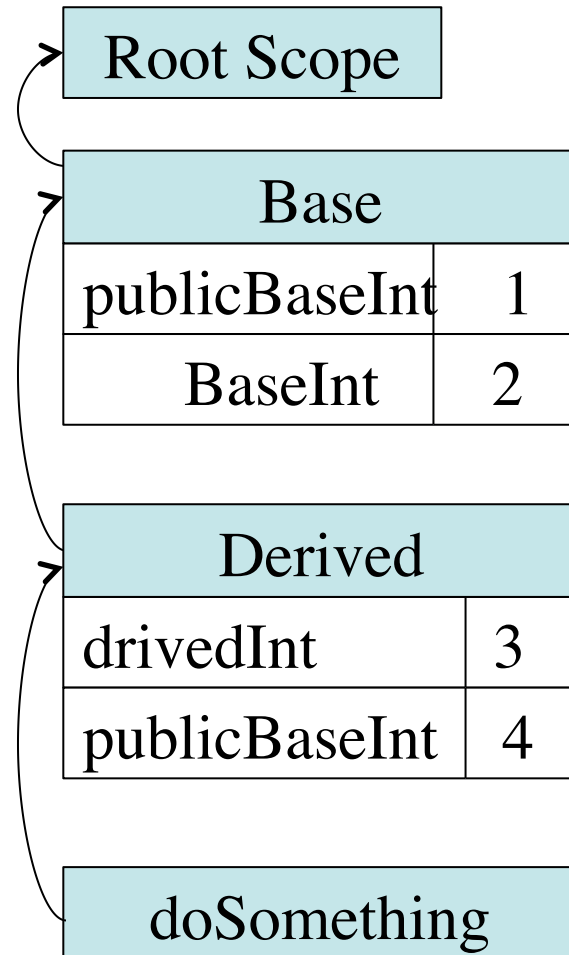
```
public class Base {  
    public int publicBaseInt = 1;  
    protected int baseInt = 2;  
}  
  
public class Derived extends Base {  
    public int derivedInt = 3;  
    public int publicBaseInt = 4;  
  
    public void doSomething () {  
        System.out.println(publicBaseInt);  
        System.out.println(baseInt);  
        System.out.println(derivedInt);  
  
        int publicBaseInt = 6;  
        System.out.println(publicBaseInt);  
    }  
}
```



Scoping with Inheritance

```
public class Base {  
    public int publicBaseInt = 1;  
    protected int baseInt = 2;  
}  
public class Derived extends Base {  
    public int derivedInt = 3;  
    public int publicBaseInt = 4;  
  
    public void doSomething () {  
        System.out.println(publicBaseInt);  
        System.out.println(baseInt);  
        System.out.println(derivedInt);  
  
        int publicBaseInt = 6;  
        System.out.println(publicBaseInt);  
    }  
}
```

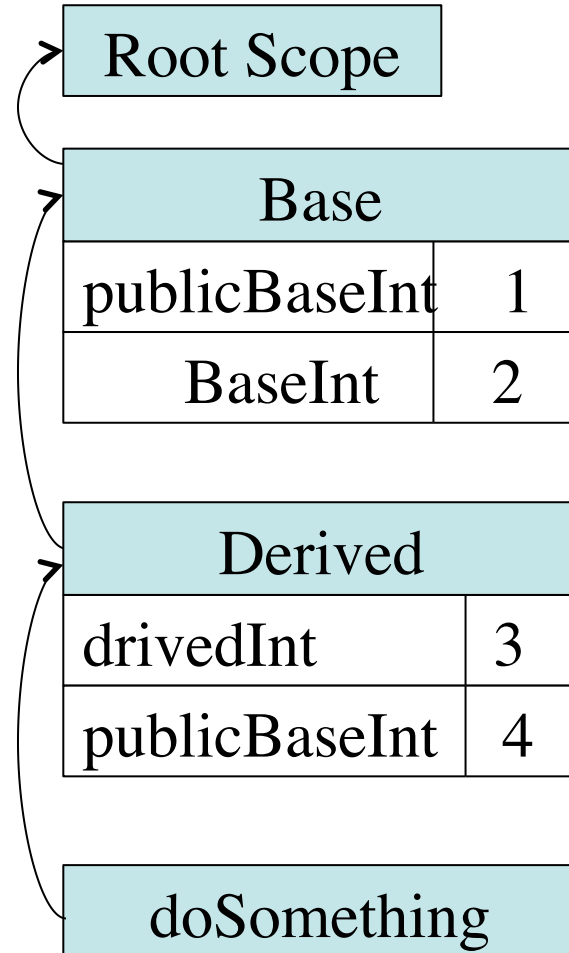
>



Scoping with Inheritance

```
public class Base {  
    public int publicBaseInt = 1;  
    protected int baseInt = 2;  
}  
public class Derived extends Base {  
    public int derivedInt = 3;  
    public int publicBaseInt = 4;  
  
    public void doSomething () {  
        System.out.println(publicBaseInt);  
        System.out.println(baseInt);  
        System.out.println(derivedInt);  
  
        int publicBaseInt = 6;  
        System.out.println(publicBaseInt);  
    }  
}
```

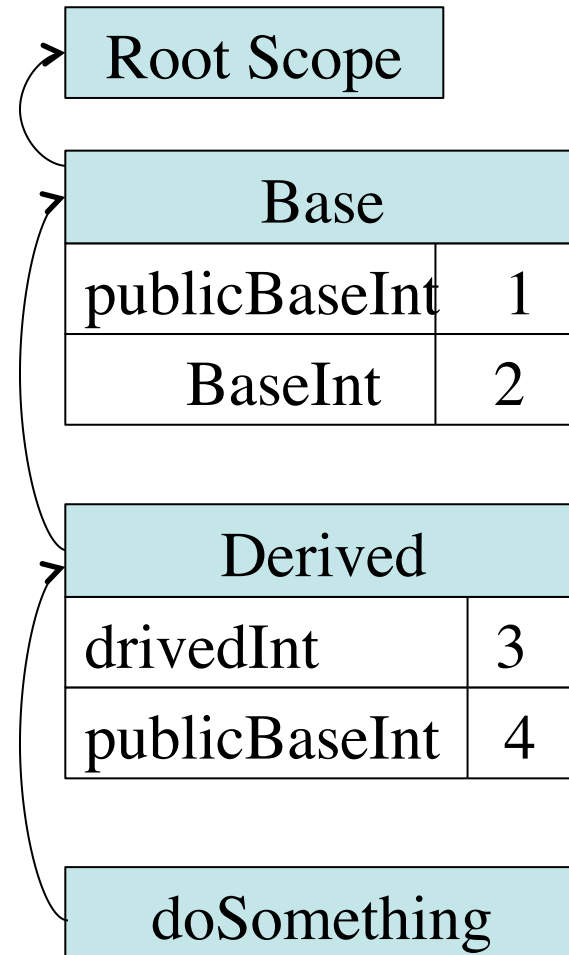
> 4



Scoping with Inheritance

```
public class Base {  
    public int publicBaseInt = 1;  
    protected int baseInt = 2;  
}  
public class Derived extends Base {  
    public int derivedInt = 3;  
    public int publicBaseInt = 4;  
  
    public void doSomething () {  
        System.out.println(publicBaseInt);  
        System.out.println(baseInt);  
        System.out.println(derivedInt);  
  
        int publicBaseInt = 6;  
        System.out.println(publicBaseInt);  
    }  
}
```

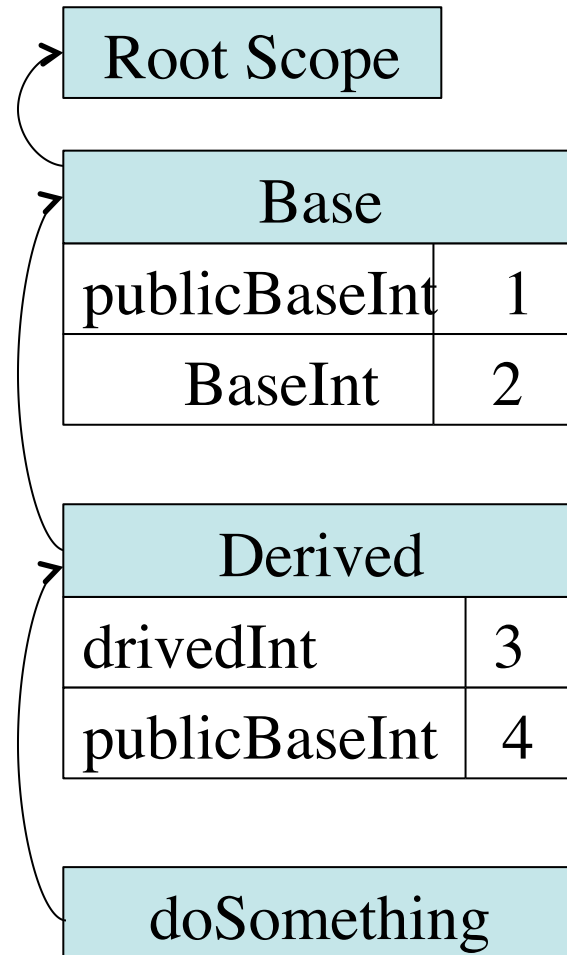
```
> 4  
2
```



Scoping with Inheritance

```
public class Base {  
    public int publicBaseInt = 1;  
    protected int baseInt = 2;  
}  
public class Derived extends Base {  
    public int derivedInt = 3;  
    public int publicBaseInt = 4;  
  
    public void doSomething () {  
        System.out.println(publicBaseInt);  
        System.out.println(baseInt);  
        System.out.println(derivedInt);  
  
        int publicBaseInt = 6;  
        System.out.println(publicBaseInt);  
    }  
}
```

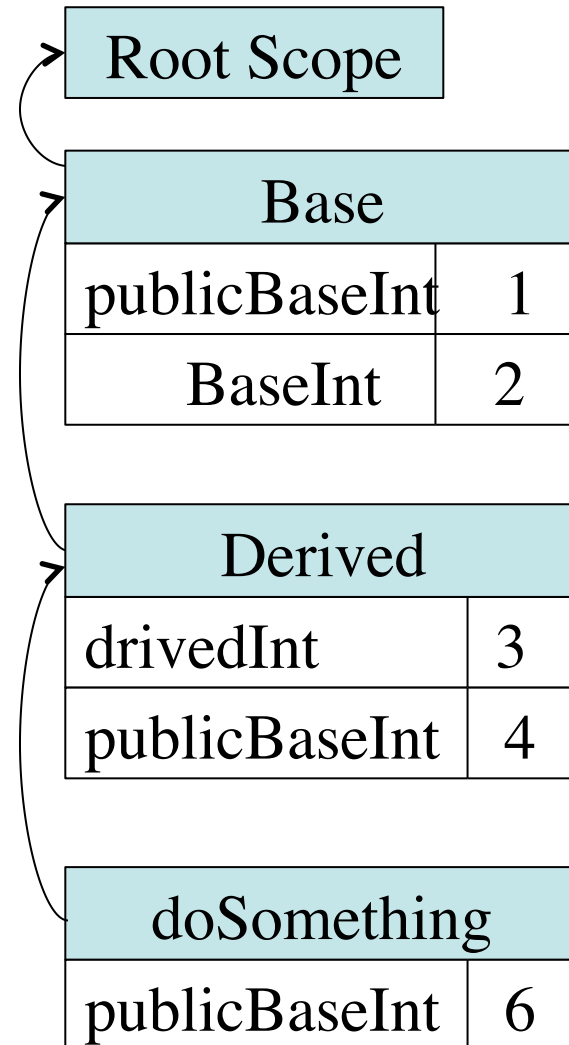
```
> 4  
2  
3
```



Scoping with Inheritance

```
public class Base {  
    public int publicBaseInt = 1;  
    protected int baseInt = 2;  
}  
public class Derived extends Base {  
    public int derivedInt = 3;  
    public int publicBaseInt = 4;  
  
    public void doSomething () {  
        System.out.println(publicBaseInt);  
        System.out.println(baseInt);  
        System.out.println(derivedInt);  
  
        int publicBaseInt = 6;  
        System.out.println(publicBaseInt);  
    }  
}
```

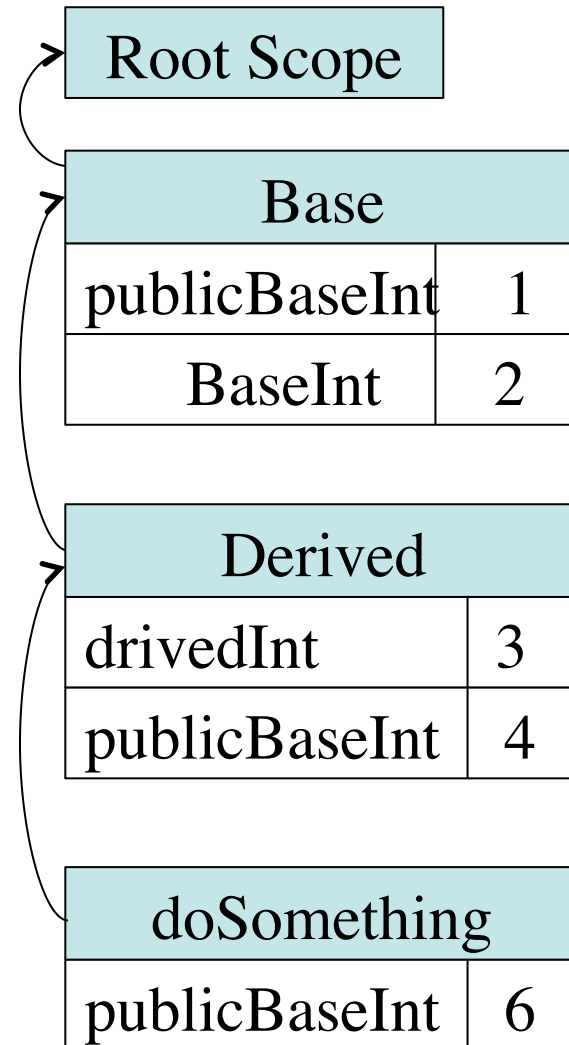
```
> 4  
2  
3
```



Scoping with Inheritance

```
public class Base {  
    public int publicBaseInt = 1;  
    protected int baseInt = 2;  
}  
  
public class Derived extends Base {  
    public int derivedInt = 3;  
    public int publicBaseInt = 4;  
  
    public void doSomething () {  
        System.out.println(publicBaseInt);  
        System.out.println(baseInt);  
        System.out.println(derivedInt);  
  
        int publicBaseInt = 6;  
        System.out.println(publicBaseInt);  
    }  
}
```

```
> 4  
2  
3  
6
```

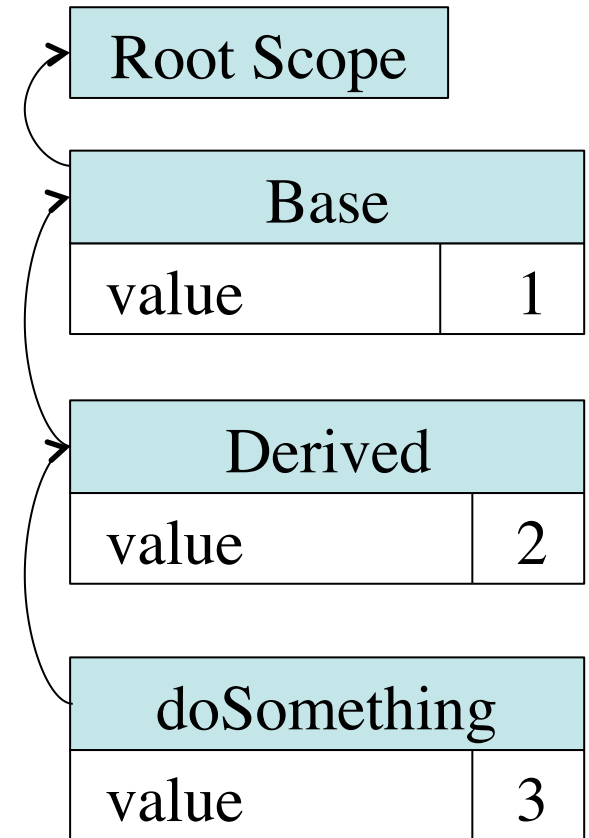


Inheritance and Scoping

- Typically, the scope for a derived class will store a link to the scope of its base class
- Looking up a field of a class traverses the scope chain until that field is found or a semantic error is found

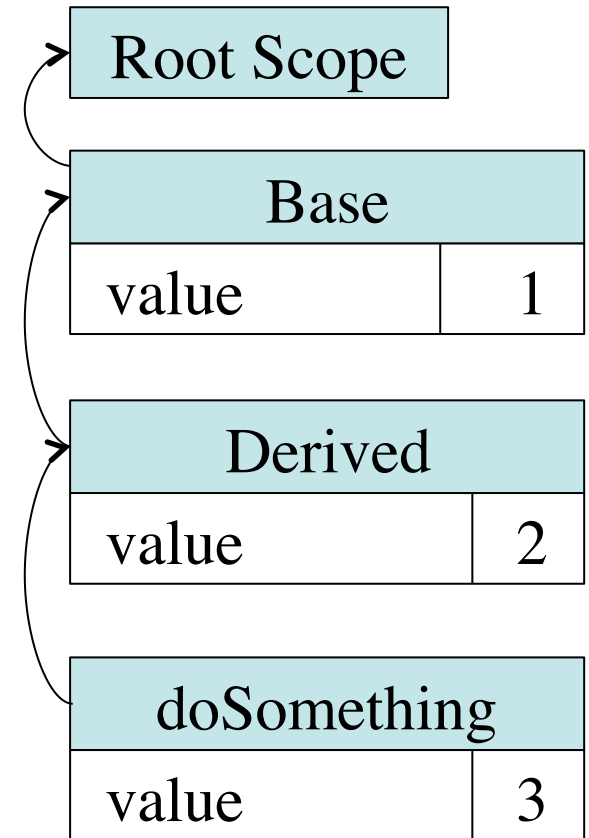
Explicit Disambiguation

```
public class Base {  
    public int value = 1;  
}  
public class Derived extends Base {  
    public int value = 2;  
  
    public void doSomething () {  
        int value = 3;  
        System.out.println(value);  
        System.out.println(this.value);  
        System.out.println(super.value);  
    }  
}
```



Explicit Disambiguation

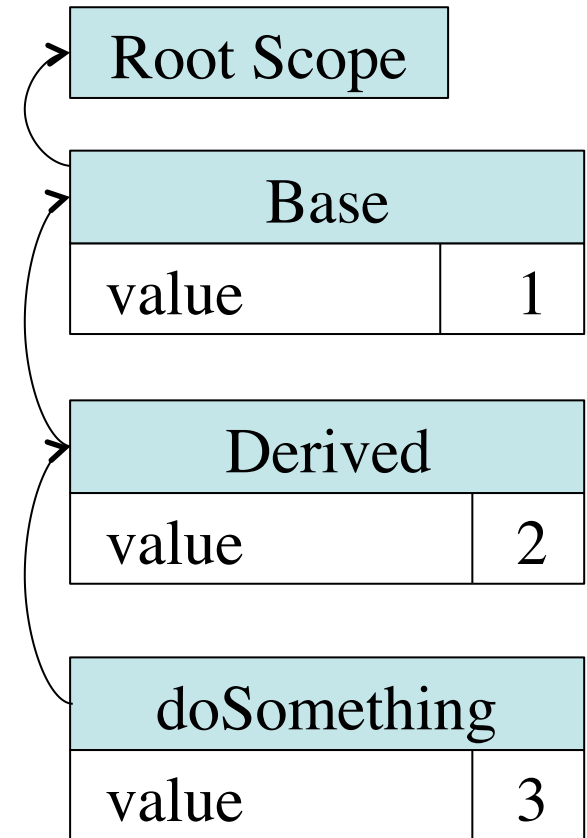
```
public class Base {  
    public int value = 1;  
}  
public class Derived extends Base {  
    public int value = 2;  
  
    public void doSomething () {  
        int value = 3;  
        System.out.println(value);  
        System.out.println(this.value);  
        System.out.println(super.value);  
    }  
}
```



Explicit Disambiguation

```
public class Base {  
    public int value = 1;  
}  
public class Derived extends Base {  
    public int value = 2;  
  
    public void doSomething () {  
        int value = 3;  
        System.out.println(value);  
        System.out.println(this.value);  
        System.out.println(super.value);  
    }  
}
```

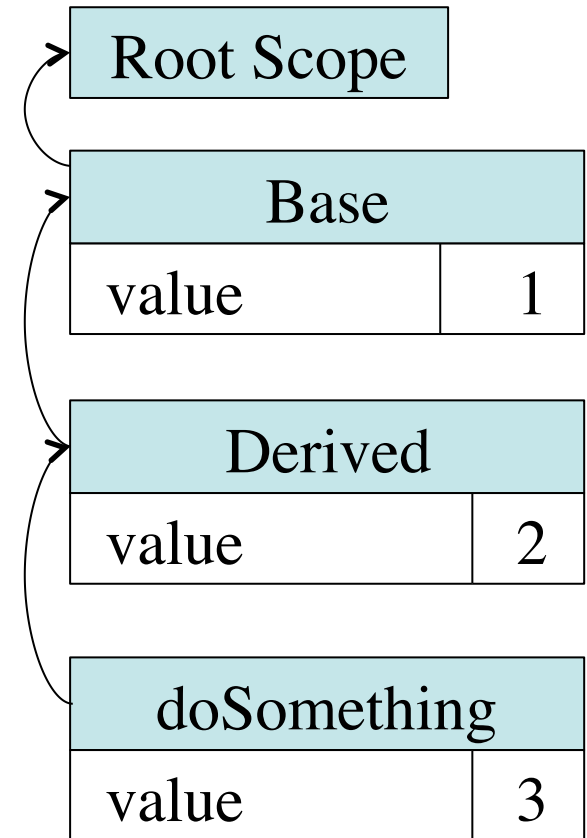
> 3



Explicit Disambiguation

```
public class Base {  
    public int value = 1;  
}  
public class Derived extends Base {  
    public int value = 2;  
  
    public void doSomething () {  
        int value = 3;  
        System.out.println(value);  
        System.out.println(this.value);  
        System.out.println(super.value);  
    }  
}
```

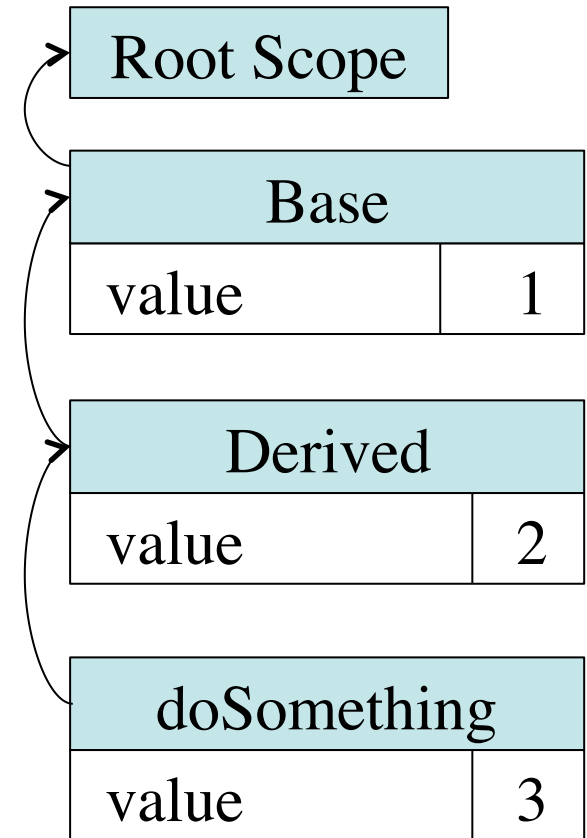
```
> 3  
  2
```



Explicit Disambiguation

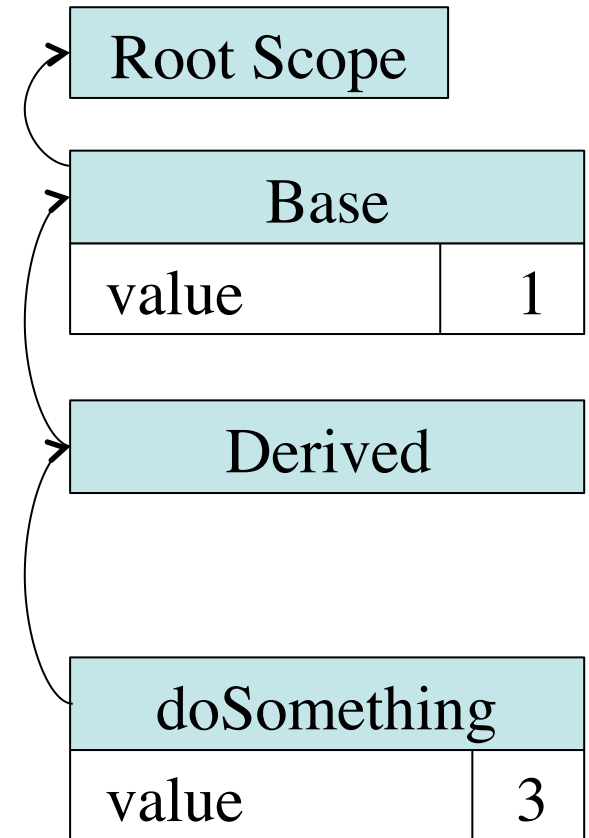
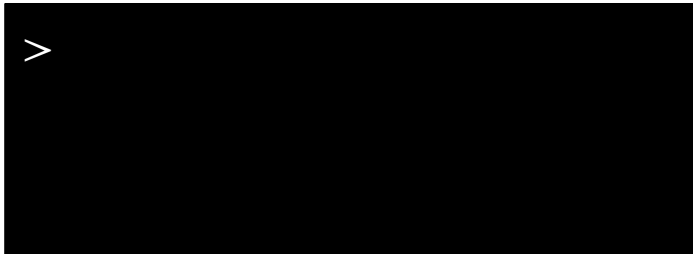
```
public class Base {  
    public int value = 1;  
}  
public class Derived extends Base {  
    public int value = 2;  
  
    public void doSomething () {  
        int value = 3;  
        System.out.println(value);  
        System.out.println(this.value);  
        System.out.println(super.value);  
    }  
}
```

```
> 3  
  2  
  1
```



Explicit Disambiguation

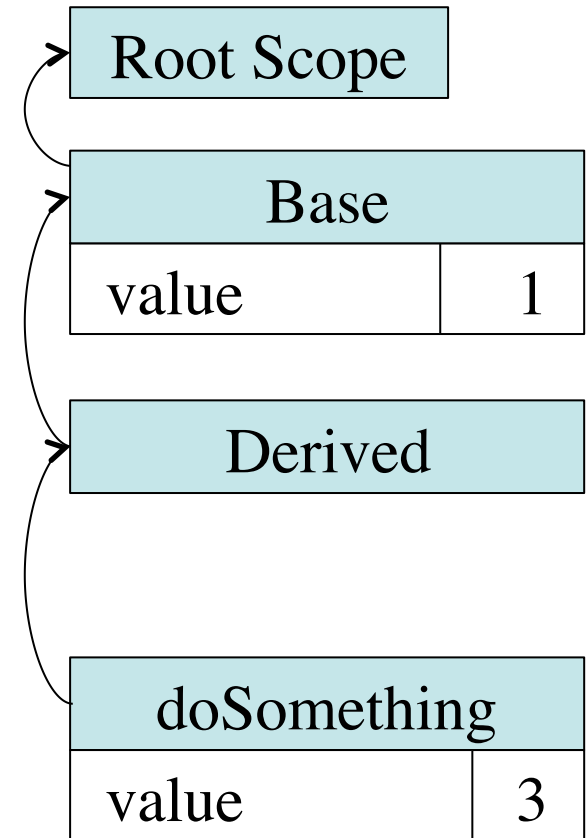
```
public class Base {  
    public int value = 1;  
}  
public class Derived extends Base {  
  
    public void doSomething () {  
        int value = 3;  
        System.out.println(value);  
        System.out.println(this.value);  
        System.out.println(super.value);  
    }  
}
```



Explicit Disambiguation

```
public class Base {  
    public int value = 1;  
}  
public class Derived extends Base {  
  
    public void doSomething () {  
        int value = 3;  
        System.out.println(value);  
        System.out.println(this.value);  
        System.out.println(super.value);  
    }  
}
```

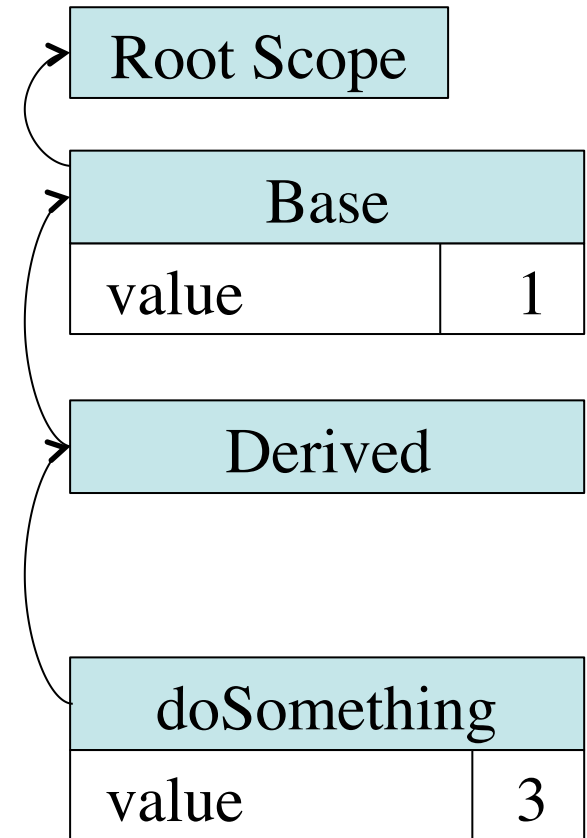
> 3



Explicit Disambiguation

```
public class Base {  
    public int value = 1;  
}  
public class Derived extends Base {  
  
    public void doSomething () {  
        int value = 3;  
        System.out.println(value);  
        System.out.println(this.value);  
        System.out.println(super.value);  
    }  
}
```

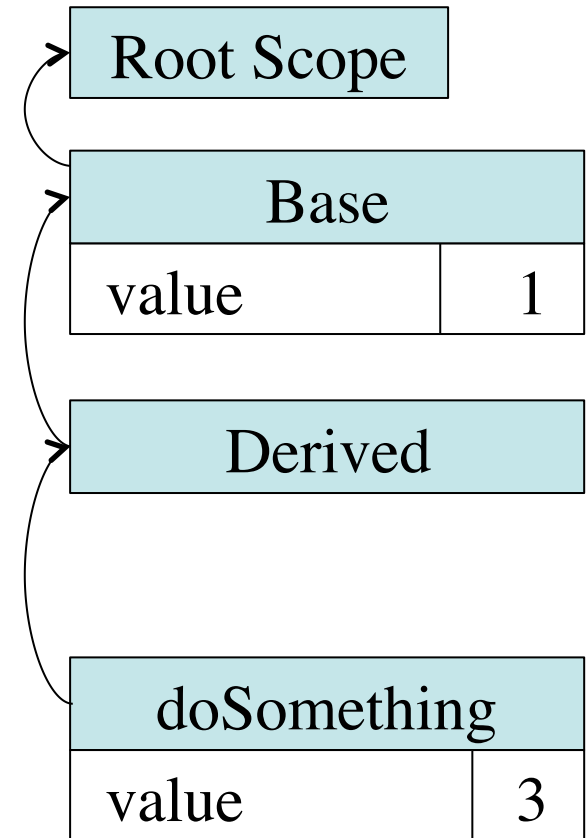
```
> 3  
1
```



Explicit Disambiguation

```
public class Base {  
    public int value = 1;  
}  
public class Derived extends Base {  
  
    public void doSomething () {  
        int value = 3;  
        System.out.println(value);  
        System.out.println(this.value);  
        System.out.println(super.value);  
    }  
}
```

```
> 3  
1  
1
```



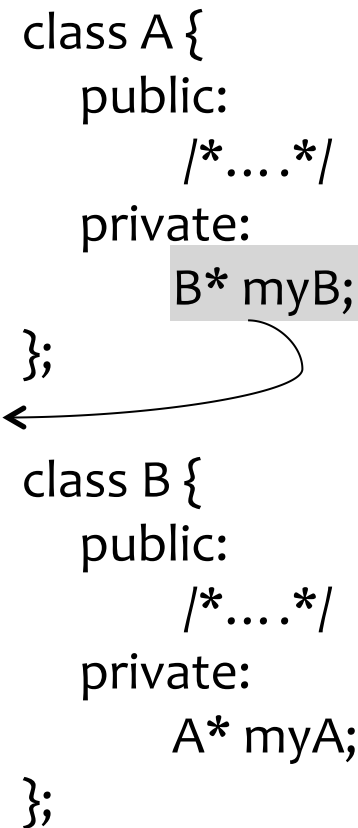
Disambiguating Scopes

- Maintain a second table of pointers into the scope stack
- When looking up a value in a specific scope, begin the search from that scope
- Some languages allow you to jump up to any arbitrary base class (for example, C++)

Scoping in C++ and Java

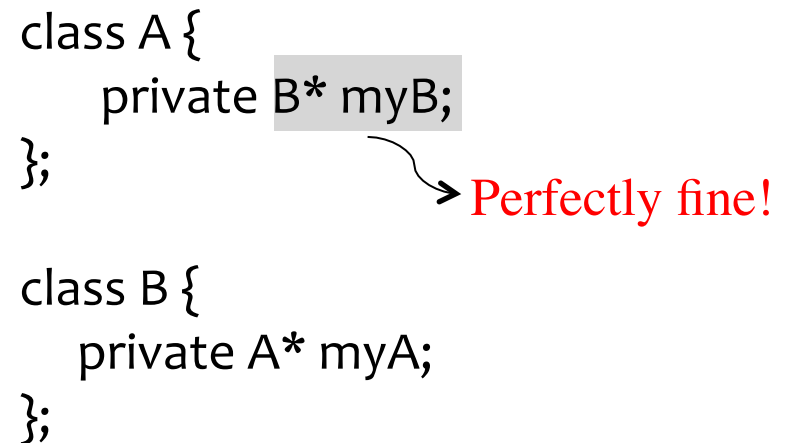
```
class A {  
    public:  
        /*...*/  
    private:  
        B* myB;  
};  
class B {  
    public:  
        /*...*/  
    private:  
        A* myA;  
};
```

Error: B not declared



```
class A {  
    private B* myB;  
};  
class B {  
    private A* myA;  
};
```

Perfectly fine!



Single and Multi-pass Compilers

- Our predictive parsing methods always scan the input from left-to-right
 - $LL(1)$, $LR(0)$, $SLR(1)$, $LALR(1)$,...
- Since we only need one token of lookahead, we can do lexical analysis and parsing simultaneously in one pass over the file
- Some compilers can combine lexical analysis, parsing, semantic analysis, and code generation into same pass
 - Single pass compilers
- Other compilers rescan the input multiple times
 - Multi-pass compilers

Single and Multi-pass Compilers

- Some languages are defined to support single-pass compilers
 - C, C++
- Some languages require multi-passes
 - Java
- Most modern compilers uses many passes over the input program

Scoping in Multi-pass Compilers

- Completely parse the input into an abstract syntax tree (first pass)
- Walk the AST, gathering information about classes (second pass)
- Walk the AST checking other properties (third pass)
- Could combine some of these

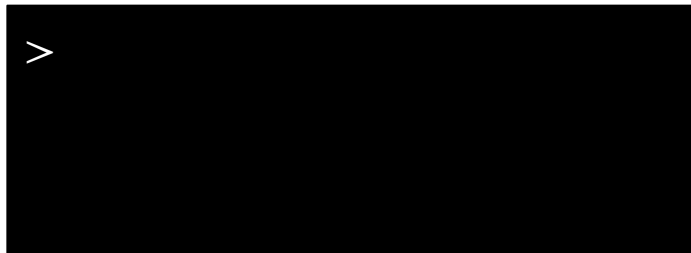
Static and Dynamic Scoping

- The scoping we've seen so far is called **static scoping** and is done at compile time
 - Identifiers refer to logically related variables
- Some languages uses **dynamic scoping**, which is done at runtime
 - Identifiers refer to the variable with that name that is closely nested at runtime

Dynamic Scoping

```
int x = 137;  
int y = 42;  
void function1 () {  
    print(x + y);  
}  
void function2 () {  
    int x = 0;  
    function1();  
}  
void function3 () {  
    int y = 0;  
    function2();  
}  
function1();  
function2();  
function3();
```

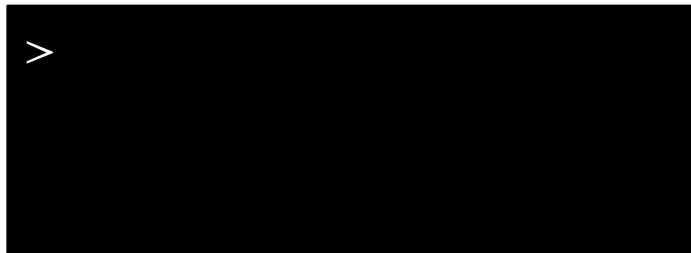
Symbol Table	
X	137
Y	42



Dynamic Scoping

```
int x = 137;  
int y = 42;  
void function1 () {  
    print(x + y);  
}  
void function2 () {  
    int x = 0;  
    function1();  
}  
void function3 () {  
    int y = 0;  
    function2();  
}  
function1();  
function2();  
function3();
```

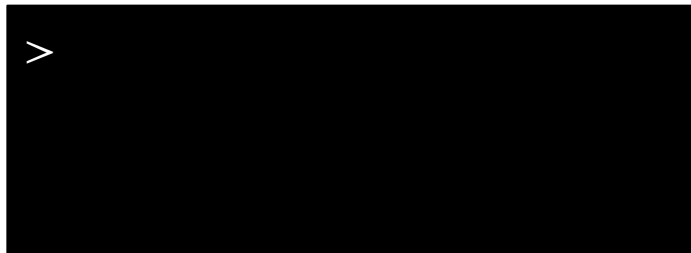
Symbol Table	
X	137
Y	42



Dynamic Scoping

```
int x = 137;  
int y = 42;  
void function1 () {  
    print(x + y);  
}  
void function2 () {  
    int x = 0;  
    function1();  
}  
void function3 () {  
    int y = 0;  
    function2();  
}  
function1();  
function2();  
function3();
```

Symbol Table	
X	137
Y	42



Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42

> 179

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42

> 179

Dynamic Scoping

```
int x = 137;  
int y = 42;  
void function1 () {  
    print(x + y);  
}  
void function2 () {  
    int x = 0;  
    function1();  
}  
void function3 () {  
    int y = 0;  
    function2();  
}  
function1();  
function2();  
function3();
```

Symbol Table	
X	137
Y	42

> 179

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42

> 179

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42
X	0

> 179

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42
X	0

> 179

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42
X	0

> 179

Dynamic Scoping

```
int x = 137;  
int y = 42;  
void function1 () {  
    print(x + y);  
}  
void function2 () {  
    int x = 0;  
    function1();  
}  
void function3 () {  
    int y = 0;  
    function2();  
}  
function1();  
function2();  
function3();
```

Symbol Table	
X	137
Y	42
X	0

> 179
42

Dynamic Scoping

```
int x = 137;  
int y = 42;  
void function1 () {  
    print(x + y);  
}  
void function2 () {  
    int x = 0;  
    function1();  
}  
void function3 () {  
    int y = 0;  
    function2();  
}  
function1();  
function2();  
function3();
```

Symbol Table	
X	137
Y	42
X	0

> 179
42

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42
X	0

> 179
42

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42

> 179
42

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42

> 179
42

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42
Y	0

> 179
42

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42
Y	0

> 179
42

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42
Y	0

> 179
42

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42
Y	0
X	0

> 179
42

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42
Y	0
X	0

> 179
42

Dynamic Scoping

```
int x = 137;  
int y = 42;  
void function1 () {  
    print(x + y);  
}  
void function2 () {  
    int x = 0;  
    function1();  
}  
void function3 () {  
    int y = 0;  
    function2();  
}  
function1();  
function2();  
function3();
```

Symbol Table	
X	137
Y	42
Y	0
X	0

> 179
42

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42
Y	0
X	0

```
> 179
42
0
```

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42
Y	0
X	0

```
> 179
42
0
```

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42
Y	0
X	0

```
> 179
42
0
```

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42
Y	0

```
> 179
42
0
```

Dynamic Scoping

```
int x = 137;
int y = 42;
void function1 () {
    print(x + y);
}
void function2 () {
    int x = 0;
    function1();
}
void function3 () {
    int y = 0;
    function2();
}
function1();
function2();
function3();
```

Symbol Table	
X	137
Y	42

```
> 179
  42
  0
```


Dynamic Scoping in Practice

- Examples: Perl
- Often implemented by preserving symbol table at runtime
- Often less efficient than static scoping
 - Compiler cannot hardcode location of variables
 - Names must be resolved at runtime

Summary

- **Semantic analysis** verifies that a syntactically valid program is correctly-formed and computes additional information about the meaning of the program
- **Scope checking** determines what objects or classes are referred to by each name in the program.
- Scope checking is usually done with a **symbol table** implemented either as a stack or **spaghetti stack**.
- In object-oriented programs, the scope for a derived class is often placed inside of the scope of a base class.
- Some semantic analyzers operate in multiple passes in order to gain more information about the program.
- In dynamic scoping, the actual execution of a program determines what each name refers to.
- With multiple inheritance, a name may need to be searched for along multiple paths.