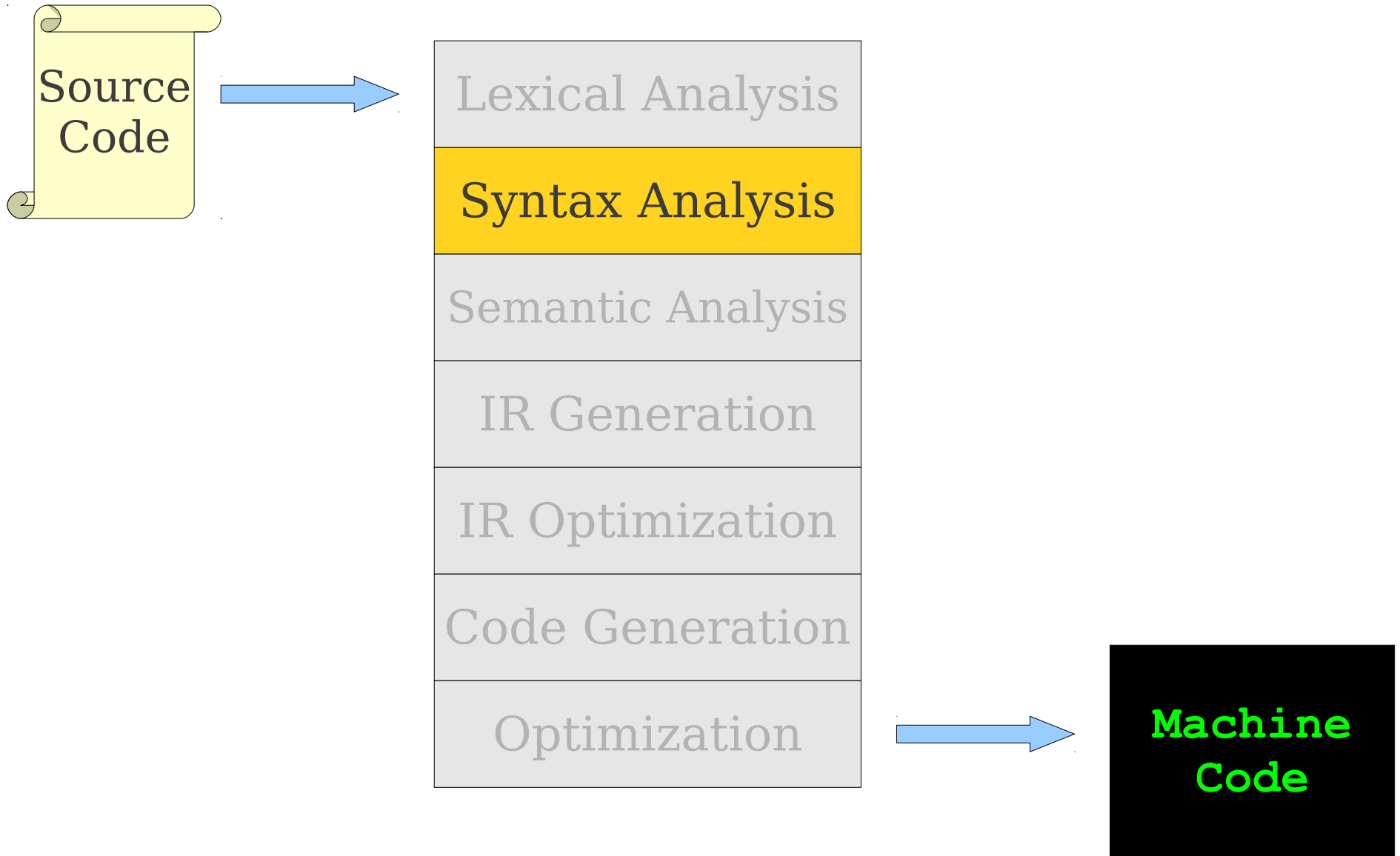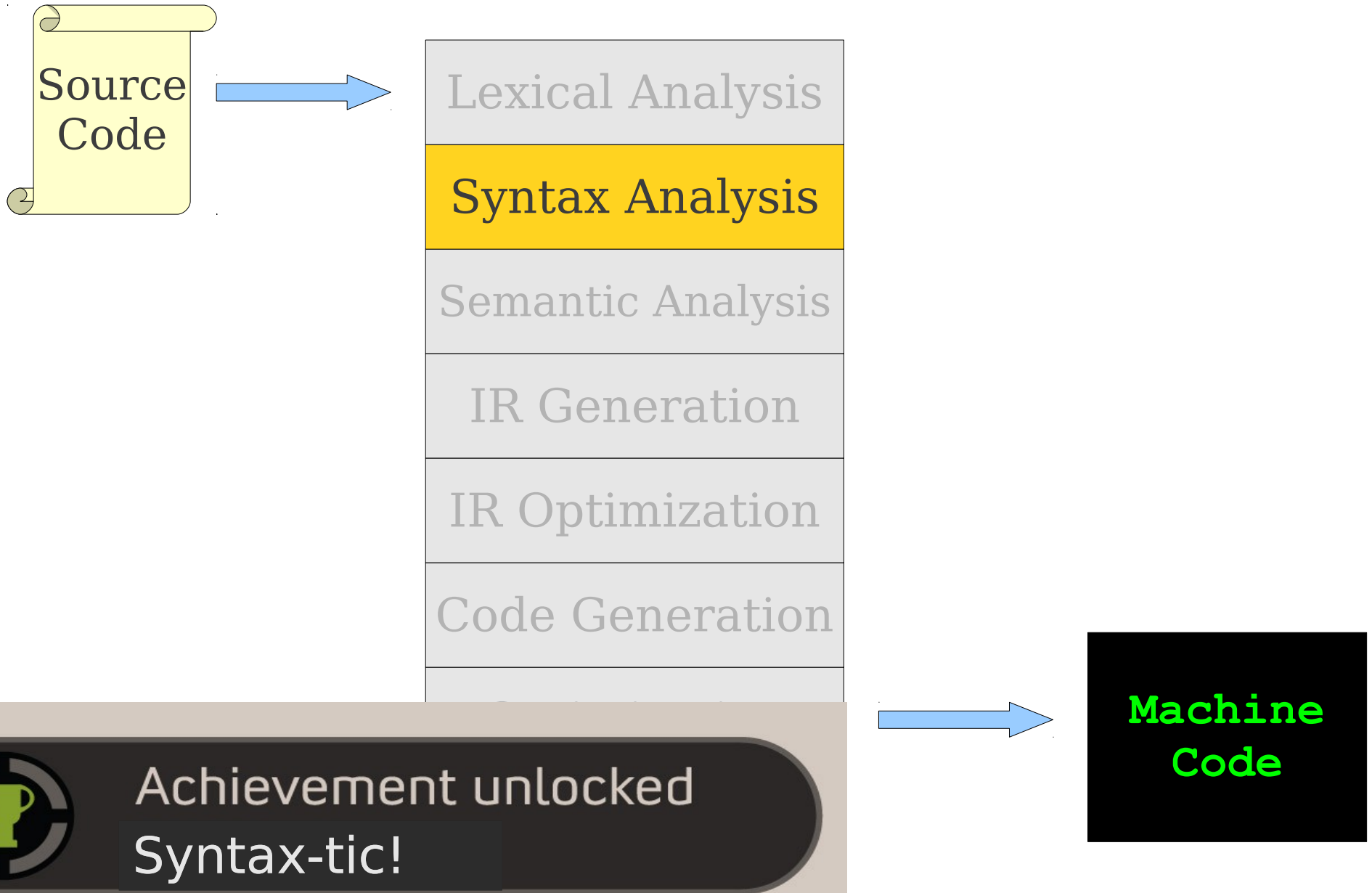# Semantic Analysis

Announcements

- Programming Project 2 due Friday at 11:59PM.

- Written Assignment 2 due Wednesday at 5:00PM.

- Questions?
  - Stop by office hours!
  - Ask on Piazza!
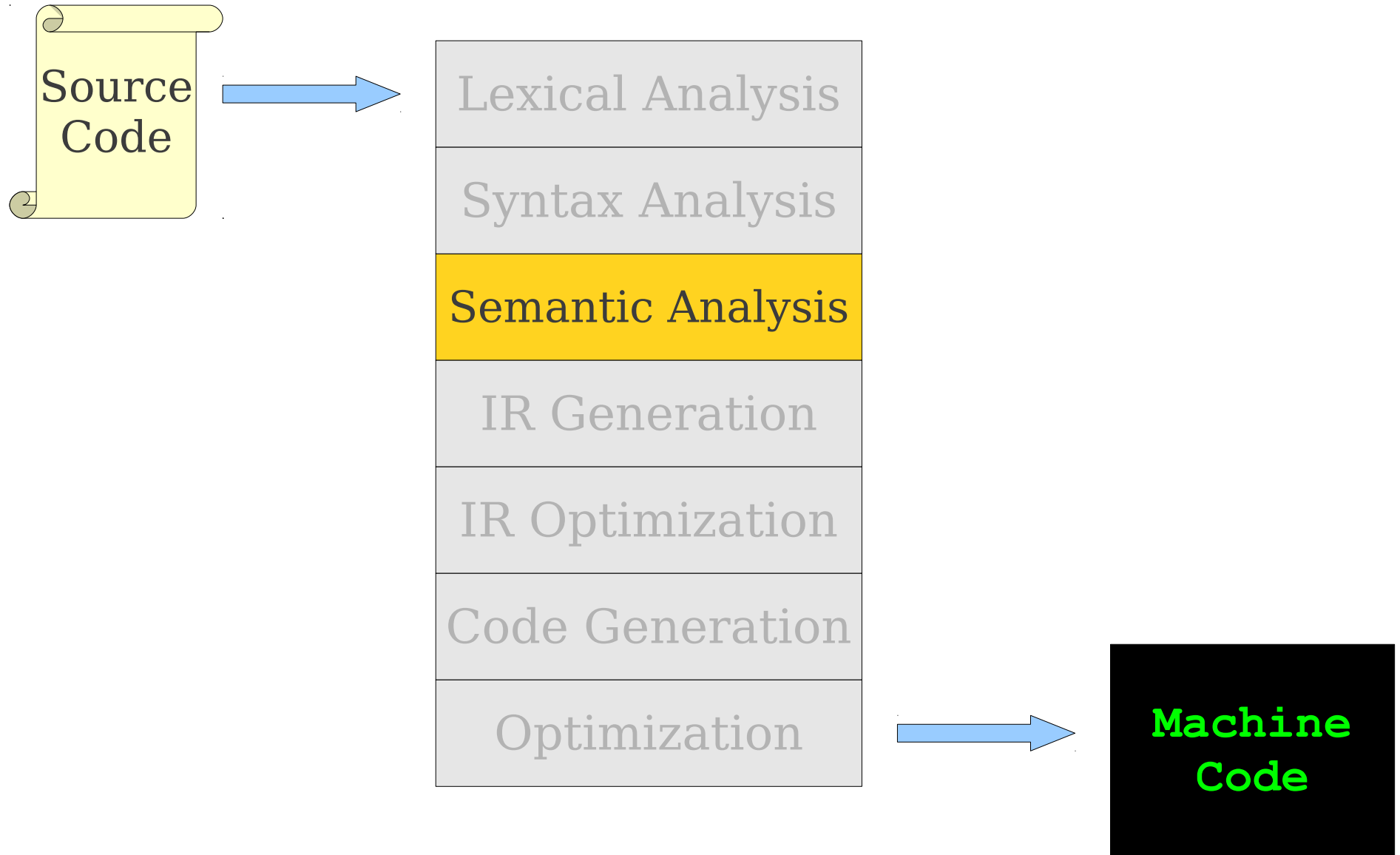 - Email the course staff!

Where We Are

Where We Are

Source
Code

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation

Machine
Code

Achievement unlocked
Syntax-tic!

Where We Are

Source Code

Lexical Analysis

Syntax Analysis

Semantic Analysis

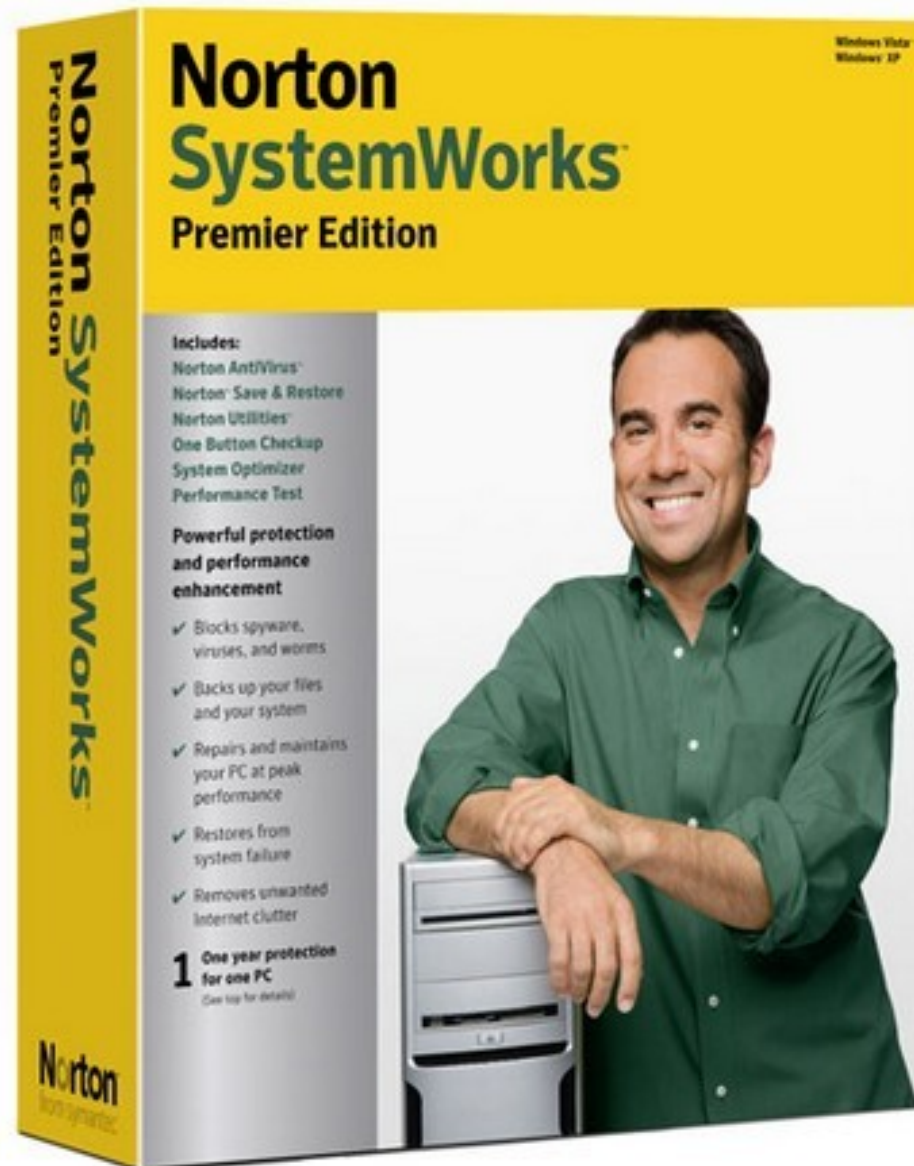IR Generation

IR Optimization

Code Generation

Optimization

Machine Code

Not Symantec Analysis

Where We Are

- Program is *lexically* well-formed:
  - Identifiers have valid names.
  - Strings are properly terminated.
  - No stray characters.
- Program is *syntactically* well-formed:
  - Class declarations have the correct structure.
  - Expressions are syntactically valid.
- Does this mean that the program is legal?

A Short Decaf Program

```
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x = new string;

        x[5] = myInteger * y;
    }
    void doSomething() {

    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
```

A Short Decaf Program

```
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x = new string;

        x[5] = myInteger * y;
    }
    void doSomething() {

    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
```

Interface not declared

Wrong type

Can't multiply strings

Variable not declared

Can't redefine functions

Can't add void

No main function

Semantic Analysis

- Ensure that the program has a well-defined **meaning**.

- Verify properties of the program that aren't caught during the earlier phases:

  - Variables are declared before they're used.

  - Expressions have the right types.

  - Arrays can only be instantiated with `NewArray`.

  - Classes don't inherit from nonexistent base classes

  - …

- Once we finish semantic analysis, we know that the user's input program is legal.

Challenges in Semantic Analysis

- Reject the largest number of incorrect programs.

- Accept the largest number of correct programs.

Validity versus Correctness

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

Validity versus Correctness

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

    }
}
```

Safe; can't
happen

Validity versus Correctness

```
int Fibonacci(int n) {
    if (n <= 1) return 0;

    return Fibonacci(n - 1) + Fibonacci(n - 2);
}

int main() {
    Print(Fibonacci(40));
}
```

Validity versus Correctness

```
int Fibonacci(int n) {
    if (n <= 1) return 0;

    return Fibonacci(n - 1) + Fibonacci(n - 2);
}

int main() {
    Print(Fibonacci(40));
}
```

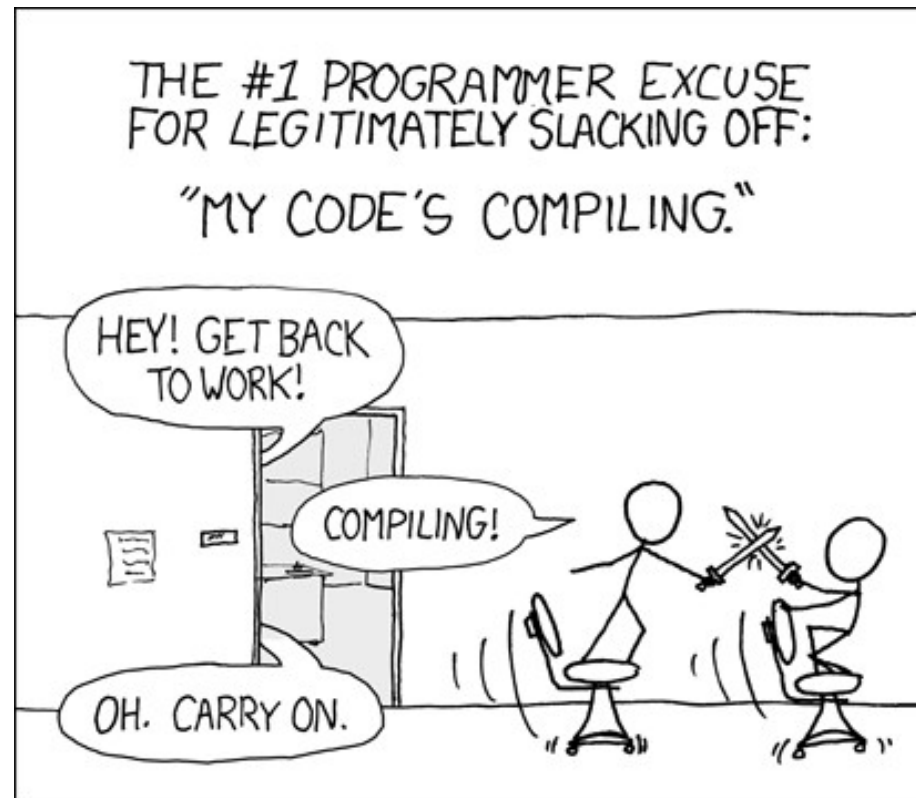*Incorrect, should be "return n;"*

Challenges in Semantic Analysis

- Reject the largest number of incorrect programs.

- Accept the largest number of correct programs.

- Do so quickly.

Challenges in Semantic Analysis

- Reject the largest number of incorrect programs.

- Accept the largest number of correct programs.

- Do so quickly.

Other Goals of Semantic Analysis

- Gather useful information about program for later phases:

  - Determine what variables are meant by each identifier.

  - Build an internal representation of inheritance hierarchies.

  - Count how many variables are in scope at each point.

Why can't we just do this during parsing?

Limitations of CFGs

- Using CFGs:
  - How would you prevent duplicate class definitions?
  - How would you differentiate variables of one type from variables of another type?
  - How would you ensure classes implement all interface methods?

Limitations of CFGs

- Using CFGs:
  - How would you prevent duplicate class definitions?
  - How would you differentiate variables of one type from variables of another type?
  - How would you ensure classes implement all interface methods?
- For most programming languages, these are *provably impossible*.
  - Use the pumping lemma for context-free languages, or Ogden's lemma.

Implementing Semantic Analysis

- **Attribute Grammars**

  - Augment `bison` rules to do checking during parsing.

  - Approach suggested in the *Compilers* book.

  - Has its limitations; more on that later.

- **Recursive AST Walk**

  - Construct the AST, then use virtual functions and recursion to explore the tree.

  - The approach we'll take in this class.

Overview for this Week

- **Scope-Checking** (Today)
  - How can we tell what object a particular identifier refers to?
  - How do we store this information?
- **Type-Checking** (Wednesday / Friday)
  - How can we tell whether expressions have valid types?
  - How do we know all function calls have valid arguments?

Scope

Not This

What's in a Name?

- The same name 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 in a program may refer to fundamentally different things:

- This is perfectly legal Java code:

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

What's in a Name?

- The same name in a program may refer to completely different objects:

- This is perfectly legal C++ code:

```cpp
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 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

- A **symbol table** is a mapping from a name to the thing that name refers to.

- As we run our semantic analysis, continuously update the symbol table with information about what is in scope.

- Questions:

  - What does this look like in practice?

  - What operations need to be defined on it?

  - How do we implement it?

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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

Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(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

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
0:  int x = 137;
1:  int z = 42;
2:  int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(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 Table Operations

- Typically implemented as a **stack of maps**.

- Each map corresponds to a particular scope.

- Stack allows for easy "enter" and "exit" operations.

- Symbol table operations are

  - **Push scope**: Enter a new scope.

  - **Pop scope**: Leave a scope, discarding all declarations in it.

  - **Insert symbol**: Add a new entry to the current scope.

  - **Lookup symbol**: Find what a name corresponds to.

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 in terms of recursive AST traversals like this.

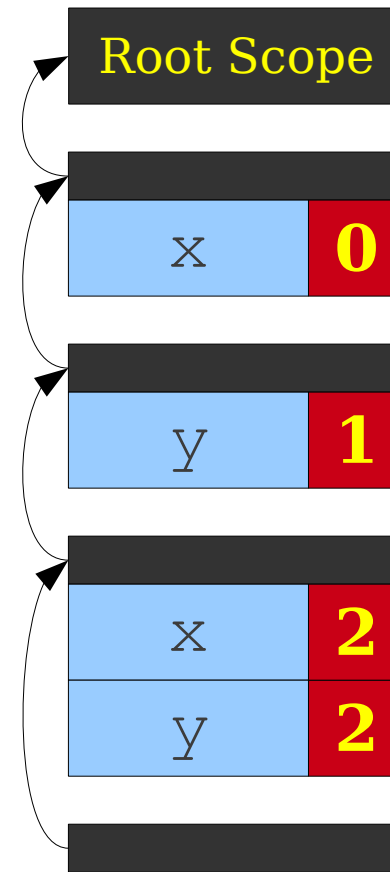# Another View of Symbol Tables
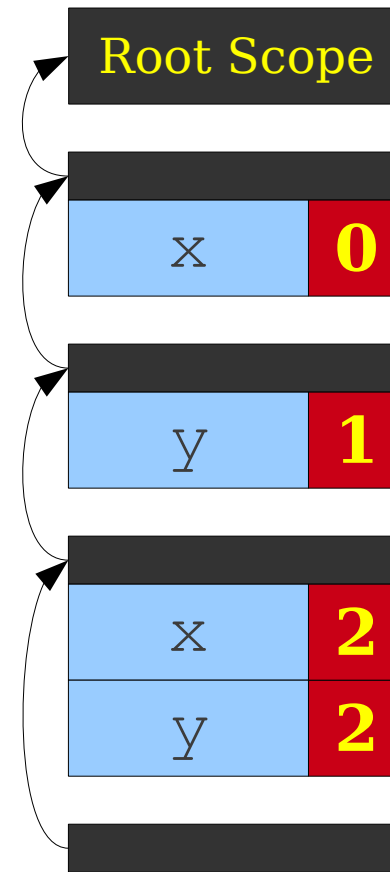
Another View of Symbol Tables

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

Another View of Symbol Tables

```
 0:  int x;
 1:  int y;
 2:  int MyFunction(int x, int y)
 3:  {
 4:     int w, z;
 5:     {
 6:        int y;
 7:     }
 8:     {
 9:        int w;
10:     }
11:  }
```
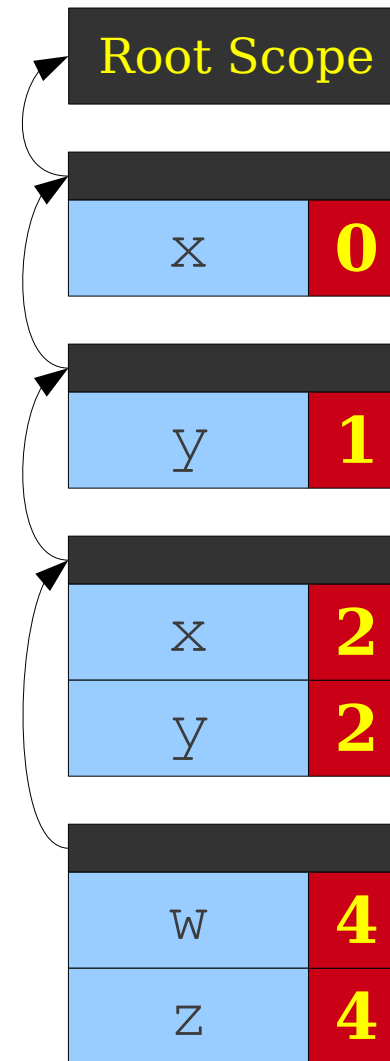
Another View of Symbol Tables

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

Another View of Symbol Tables

Root Scope

X	**0**

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```
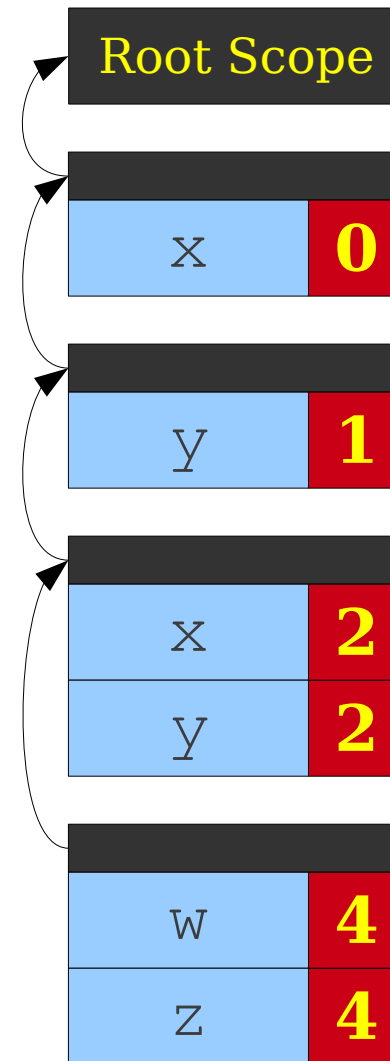
Another View of Symbol Tables

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

Root Scope

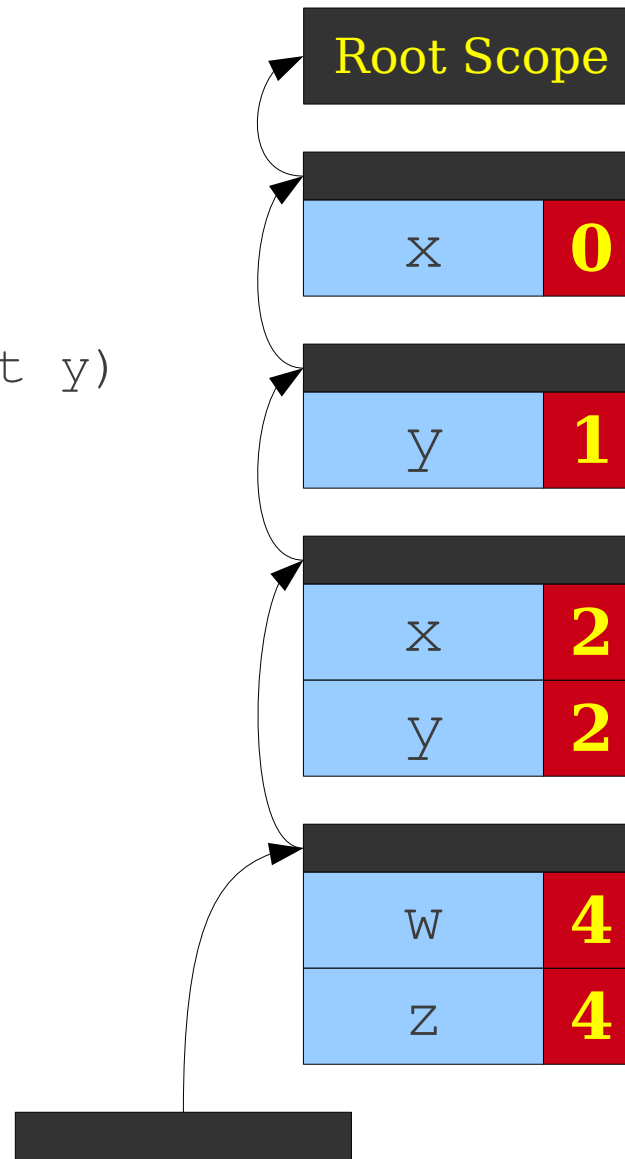| X | 0 |

Another View of Symbol Tables

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

Another View of Symbol Tables

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

Root Scope

x	**0**

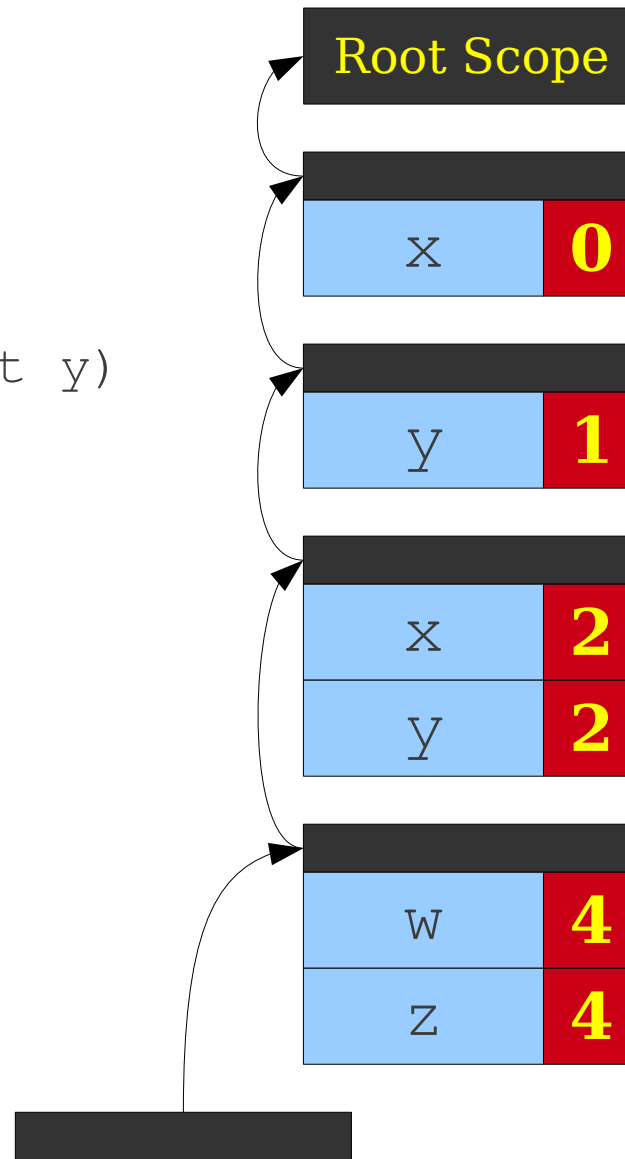| y	**1**

Another View of Symbol Tables

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

Another View of Symbol Tables

```
 0:  int x;
 1:  int y;
 2:  int MyFunction(int x, int y)
 3:  {
 4:     int w, z;
 5:     {
 6:        int y;
 7:     }
 8:     {
 9:        int w;
10:     }
11:  }
```
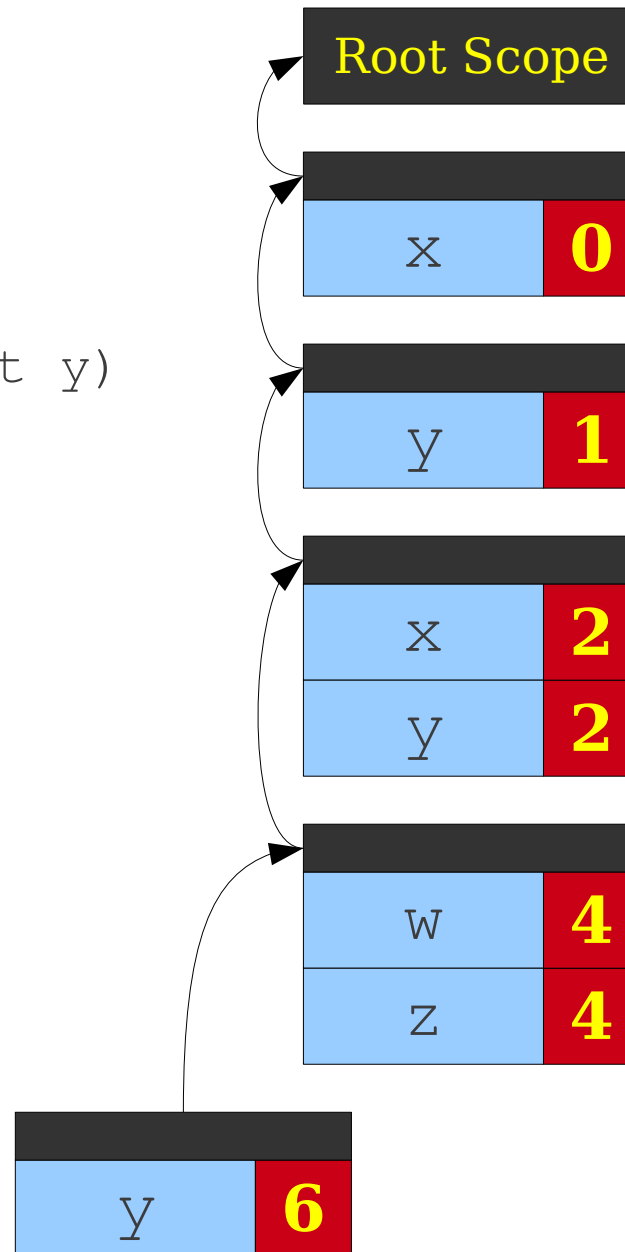
Another View of Symbol Tables

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


Root Scope

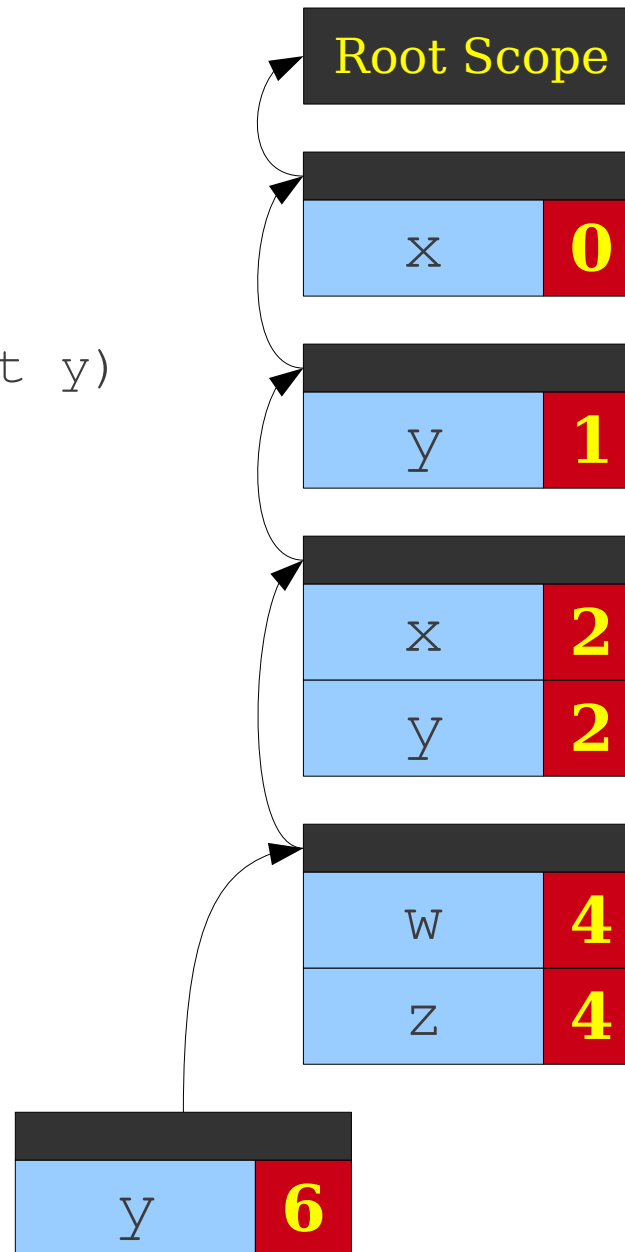| x | 0 |
y	1
x	2
y	2

Another View of Symbol Tables

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

Another View of Symbol Tables

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


Root Scope

| x | 0 |
| y | 1 |

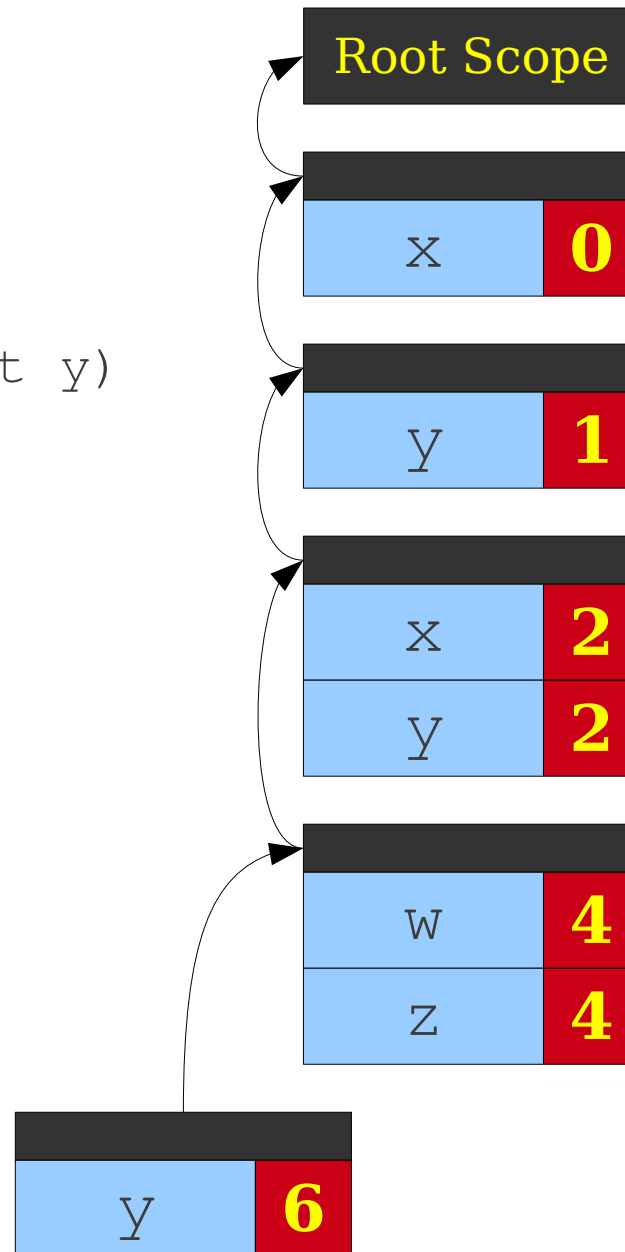| x | 2 |
| y | 2 |

| w | 4 |
| z | 4 |

Another View of Symbol Tables

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

Root Scope

x	0

y	1

x	2
y	2

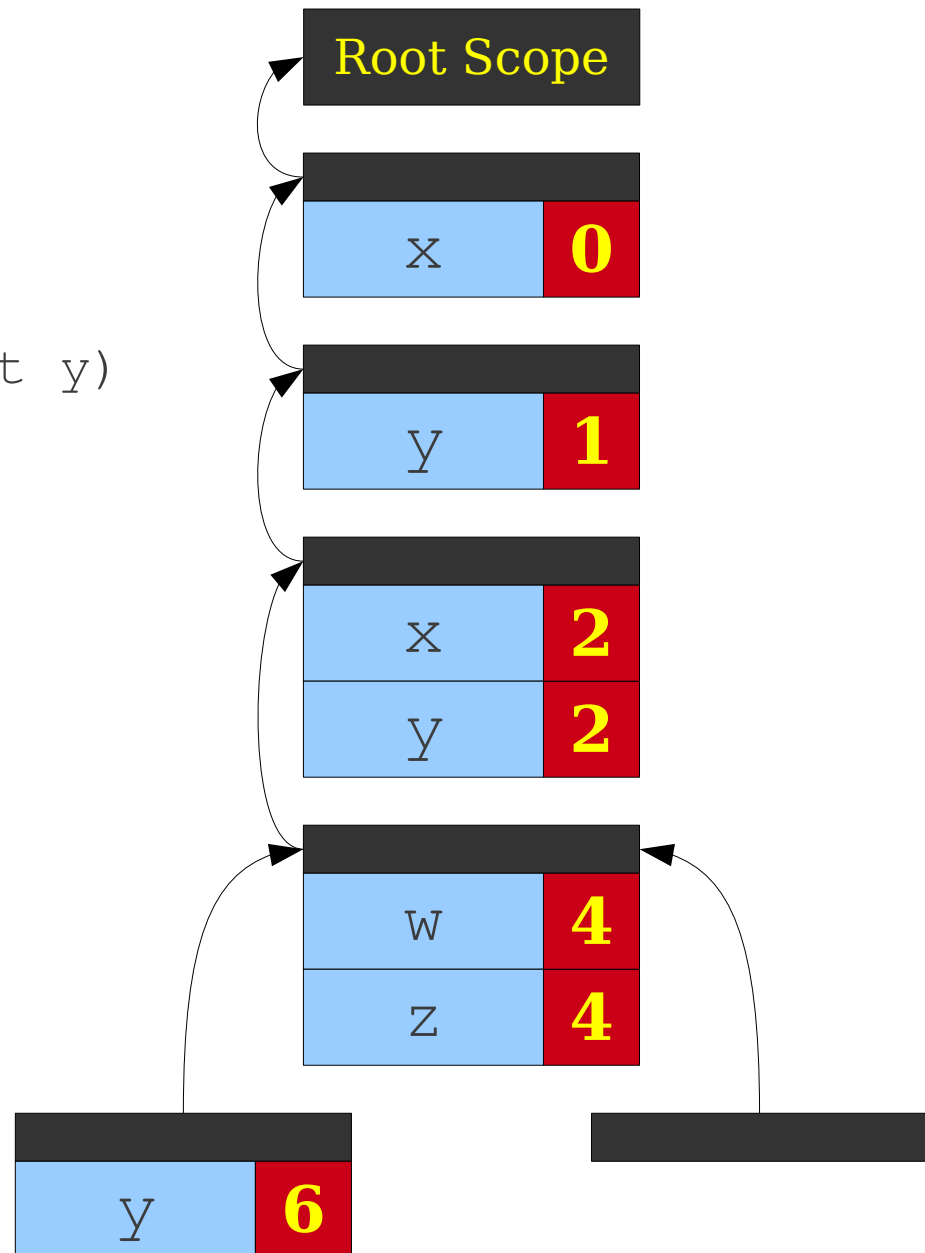| w	4
z	4

Another View of Symbol Tables

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

Root Scope

x	**0**

y	**1**

x	**2**
y	**2**

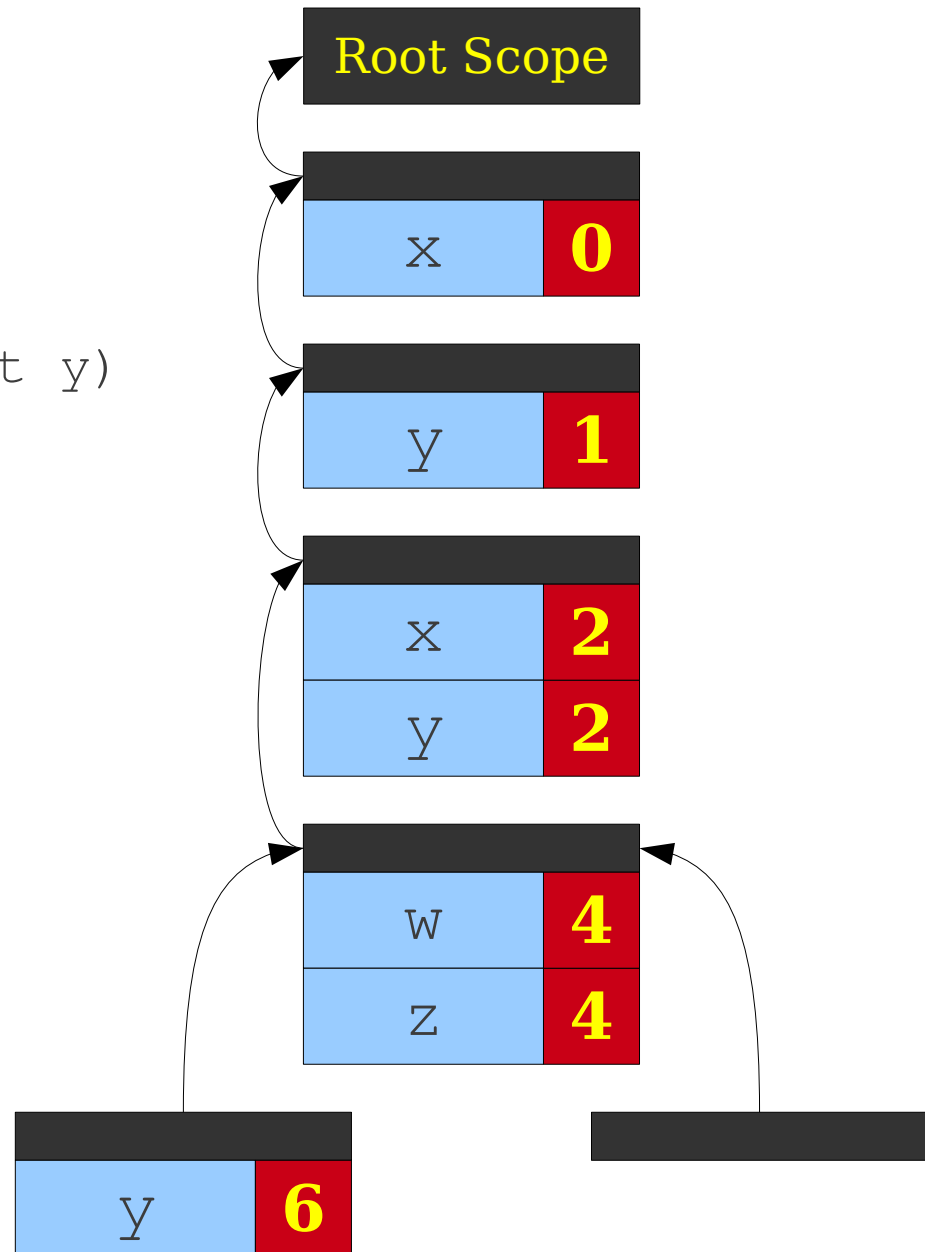|	
w	**4**
z	**4**

Another View of Symbol Tables

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

Root Scope

| x | 0 |

| y | 1 |

| x | 2 |
| y | 2 |

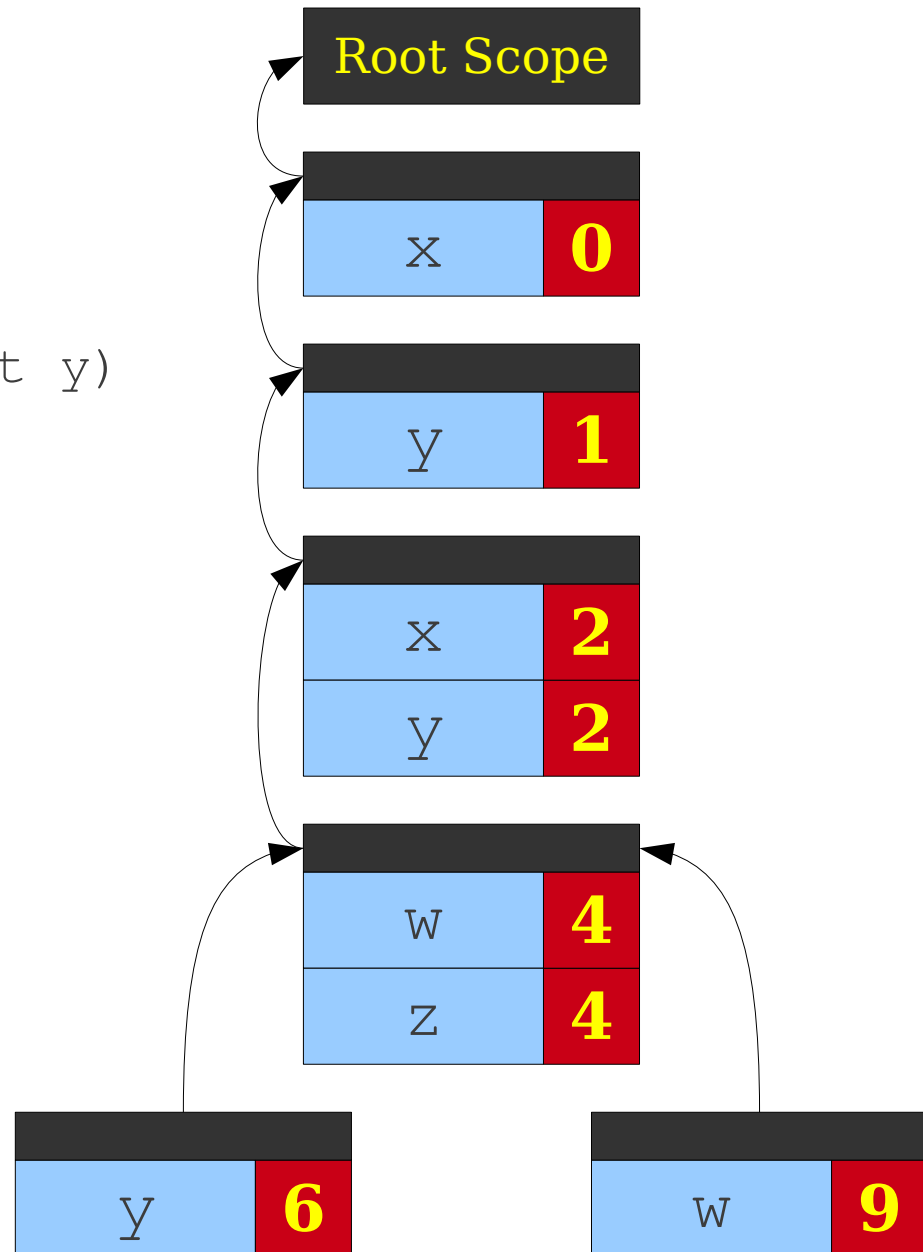| w | 4 |
| z | 4 |

Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:    int w, z;
 5:    {
 6:      int y;
 7:    }
 8:    {
 9:      int w;
10:    }
11: }
```
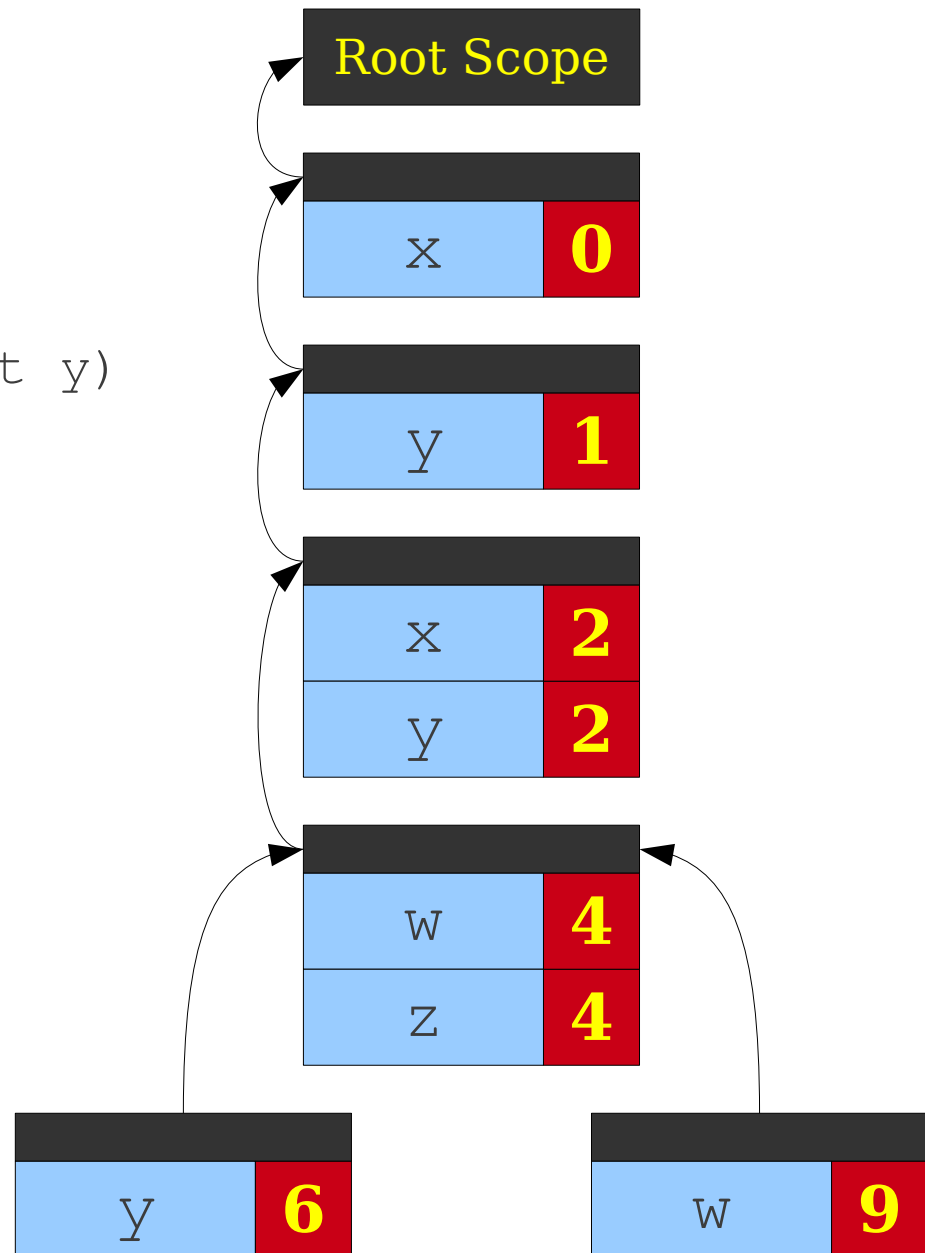
Another View of Symbol Tables

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

Root Scope

x	**0**

y	**1**

x	**2**
y	**2**

w	**4**
z	**4**

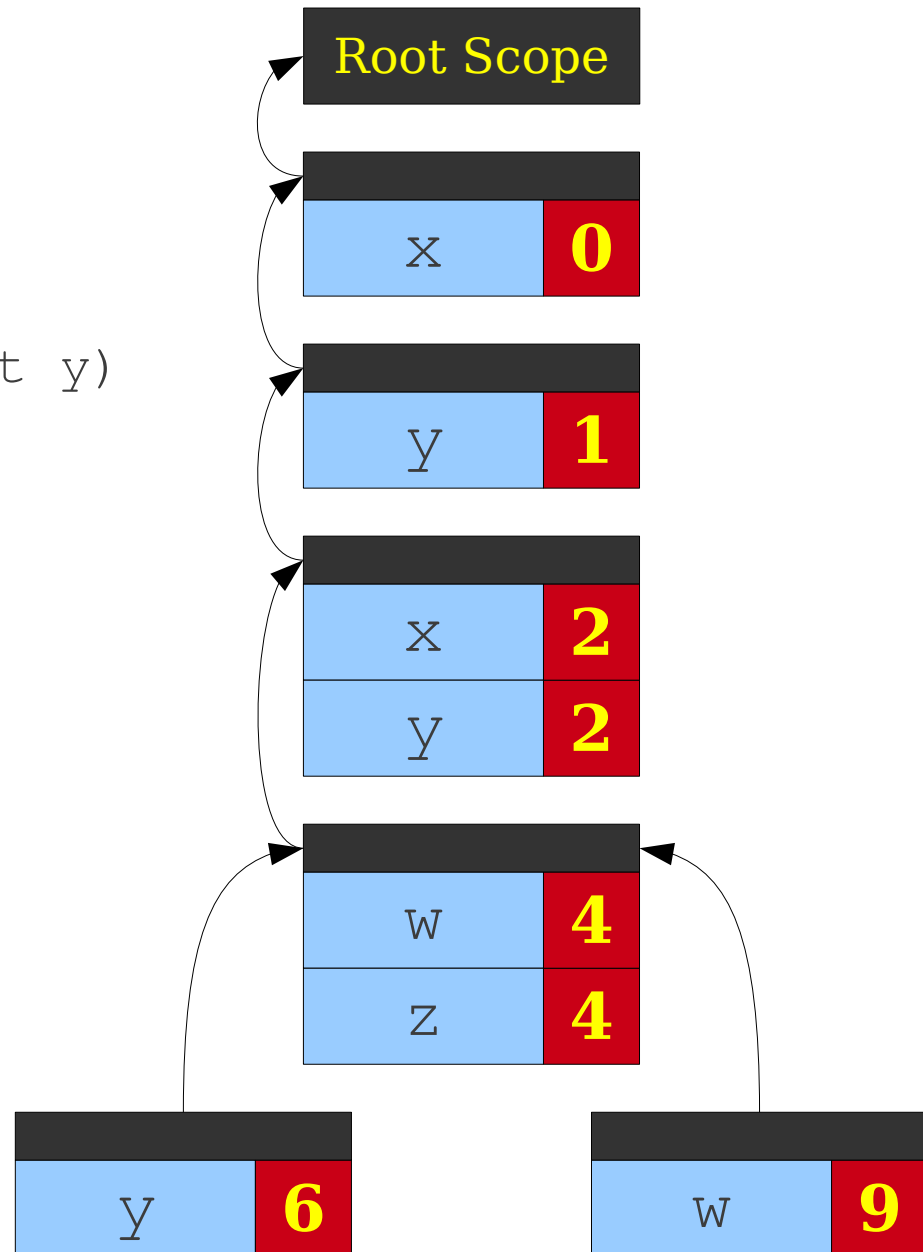| y	**6**

Another View of Symbol Tables

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

Root Scope

| x | **0** |

| y | **1** |

| x | **2** |
| y | **2** |

| w | **4** |
| z | **4** |

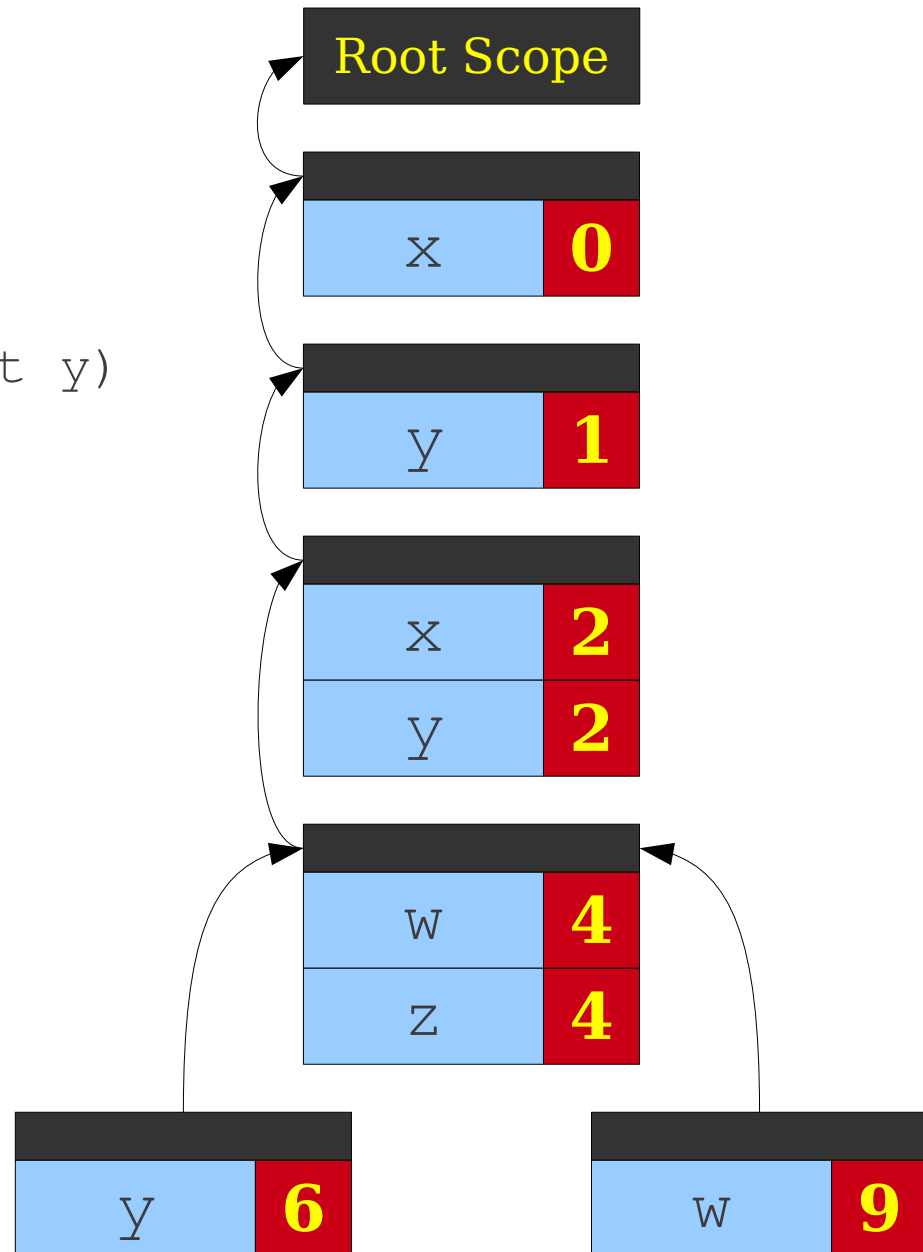| y | **6** |

Another View of Symbol Tables

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

Root Scope

x	0

y	1

x	2
y	2

w	4
z	4

y	6

Another View of Symbol Tables

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

Another View of Symbol Tables

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

Another View of Symbol Tables

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

Root Scope

| x | **0** |

| y | **1** |

| x | **2** |
| y | **2** |

| w | **4** |
| z | **4** |

| y | **6** |

| w | **9** |

Another View of Symbol Tables

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

Root Scope

x	**0**

y	**1**

x	**2**
y	**2**

w	**4**
z	**4**

y	**6**

w	**9**

Another View of Symbol Tables

```
0:  int x;
1:  int y;
2:  int MyFunction(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 parents, 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 more accurately captures the scoping structure.

- Spaghetti stack is a *static* structure; explicit stack is a *dynamic* structure.

- Explicit stack is an optimization of a spaghetti stack; more on that later.

Scoping in Object-Oriented Languages

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;
}
```

Scoping with Inheritance

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

Root Scope	

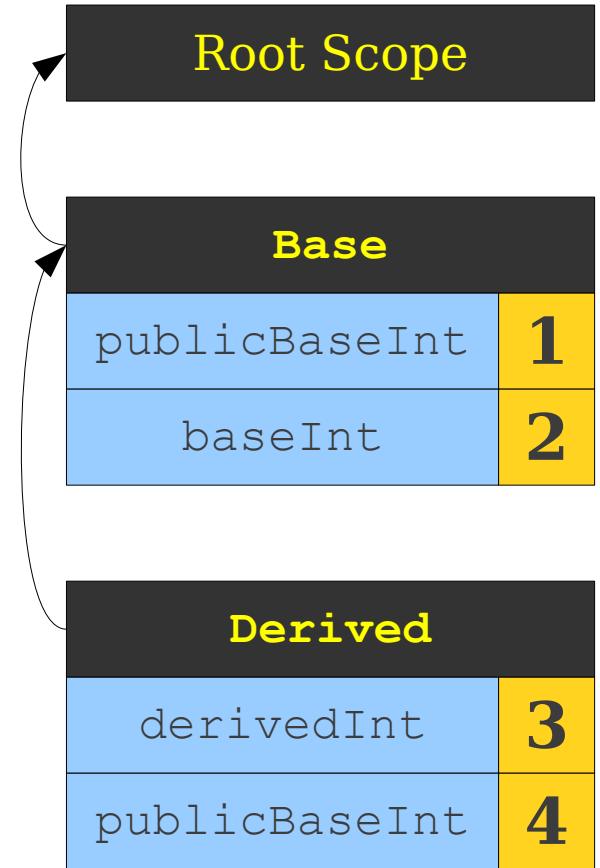| Base	
publicBaseInt	1
baseInt	2

Scoping with Inheritance

```java
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);
    }
}
```
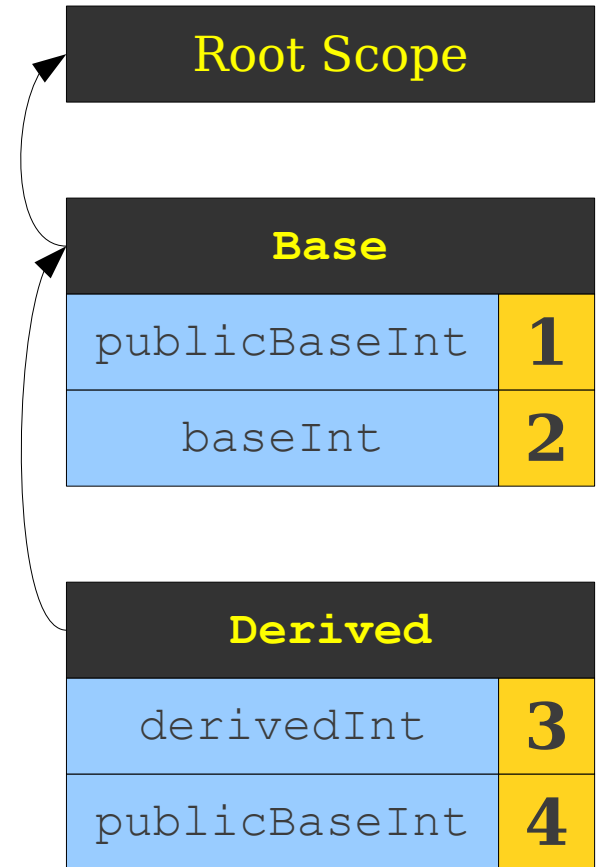
Scoping with Inheritance

```java
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

```java
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);
    }
}
```


Root Scope

Base	
publicBaseInt	1
baseInt	2

Derived	
derivedInt	3
publicBaseInt	4

```
>
```
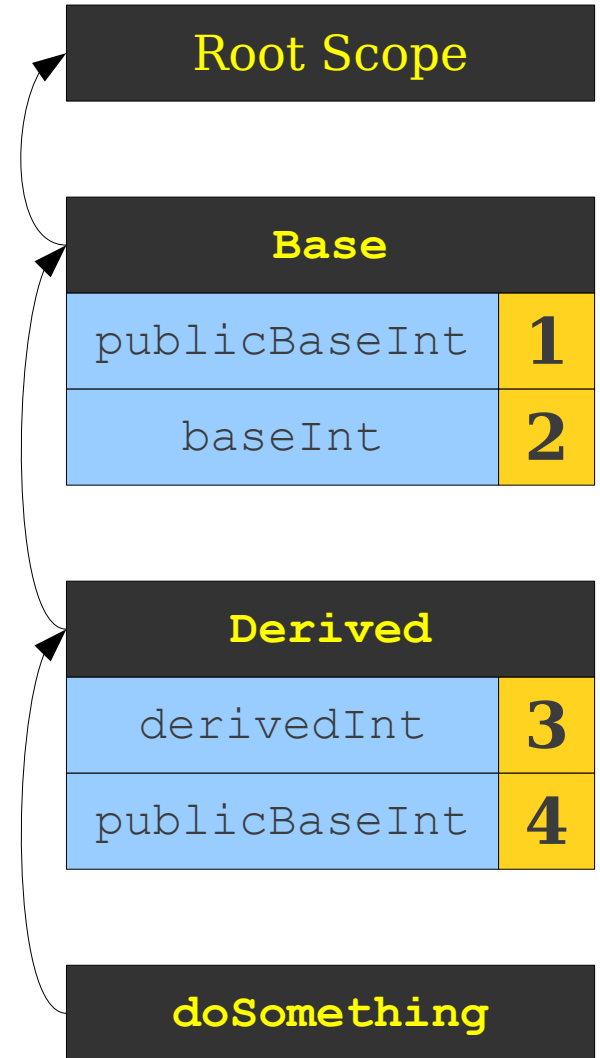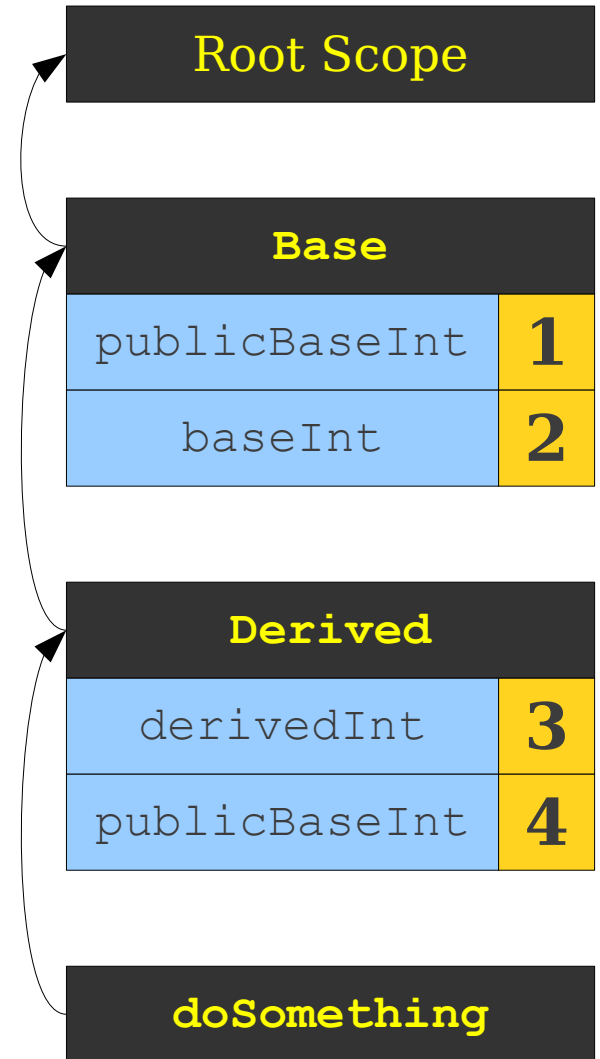
Scoping with Inheritance

```java
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);
    }
}
```

Root Scope	

Base	
publicBaseInt	1
baseInt	2

Derived	
derivedInt	3
publicBaseInt	4

```
>
```
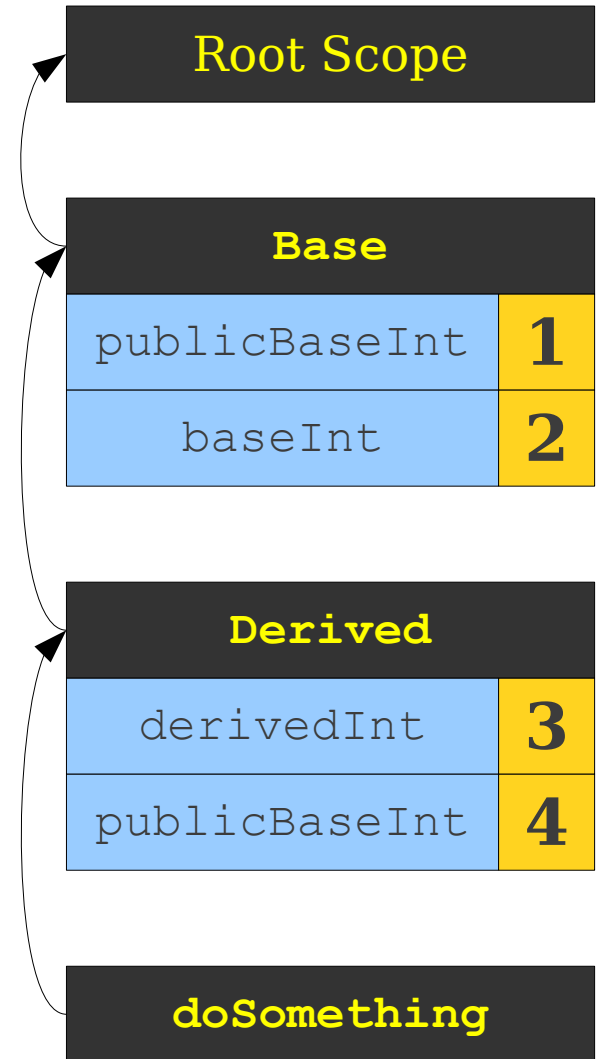
Scoping with Inheritance

```java
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);
    }
}
```

```
>
```

Root Scope

Base

publicBaseInt	1
baseInt	2

Derived

derivedInt	3
publicBaseInt	4

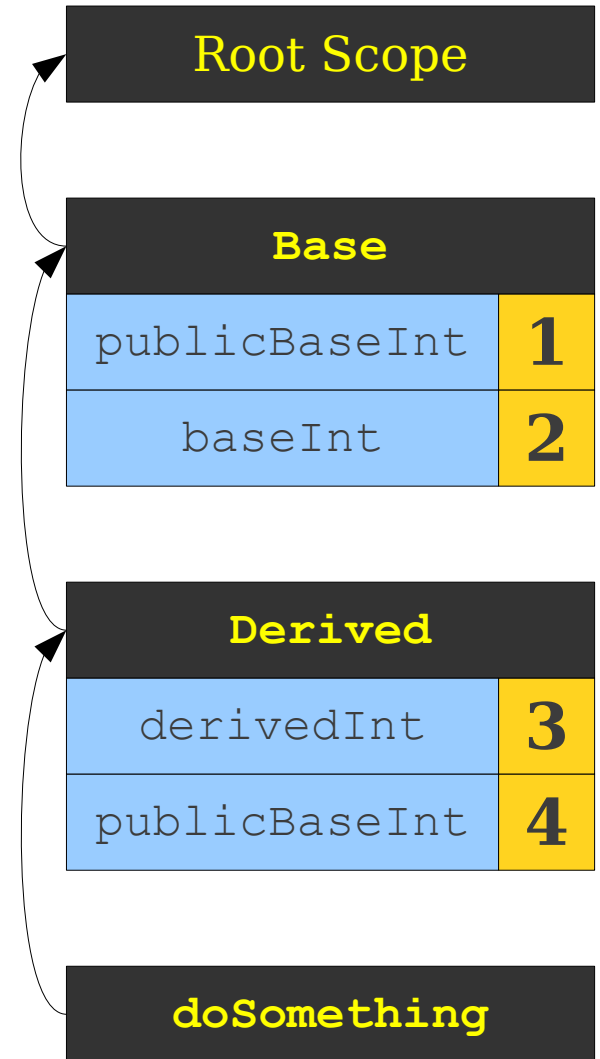**doSomething**

Scoping with Inheritance

```java
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);
    }
}
```

>

Root Scope

Base

publicBaseInt	1
baseInt	2

Derived

derivedInt	3
publicBaseInt	4

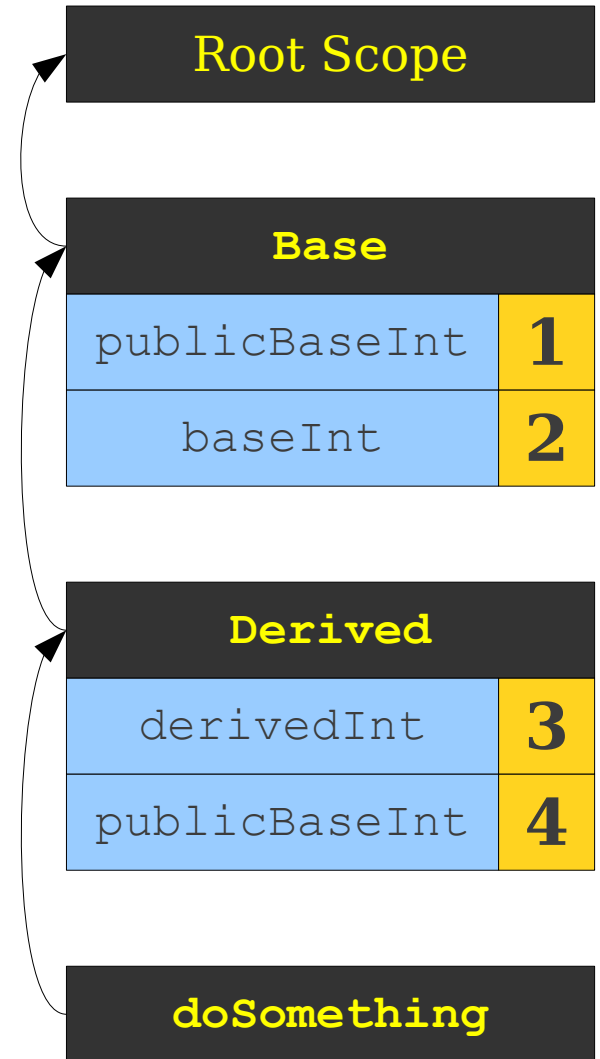**doSomething**

Scoping with Inheritance

```java
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
```


Root Scope

Base

| publicBaseInt | 1 |
| baseInt | 2 |

Derived

| derivedInt | 3 |
| publicBaseInt | 4 |

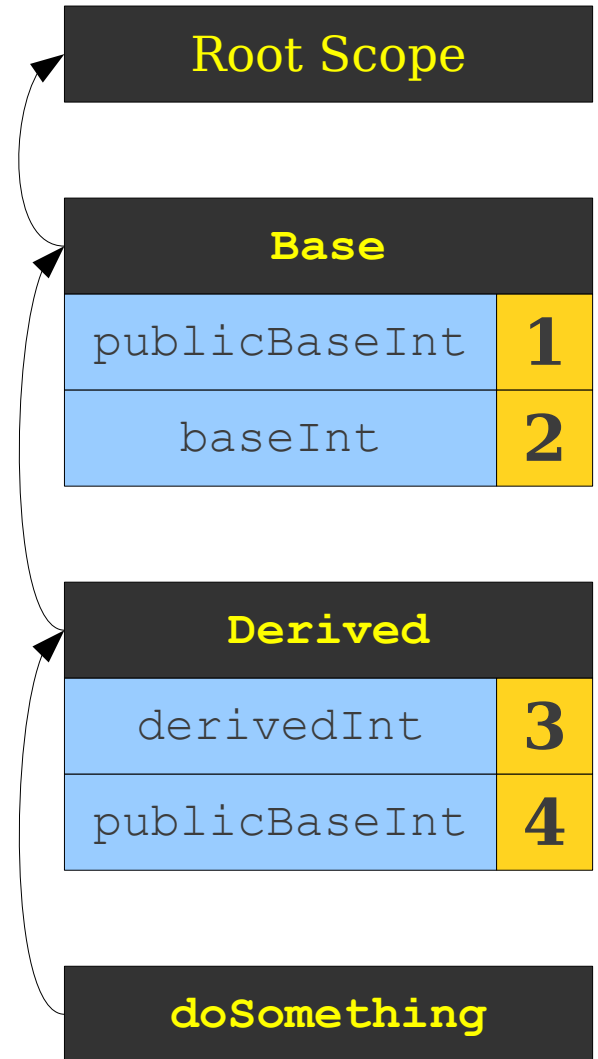**doSomething**

Scoping with Inheritance

```java
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
```
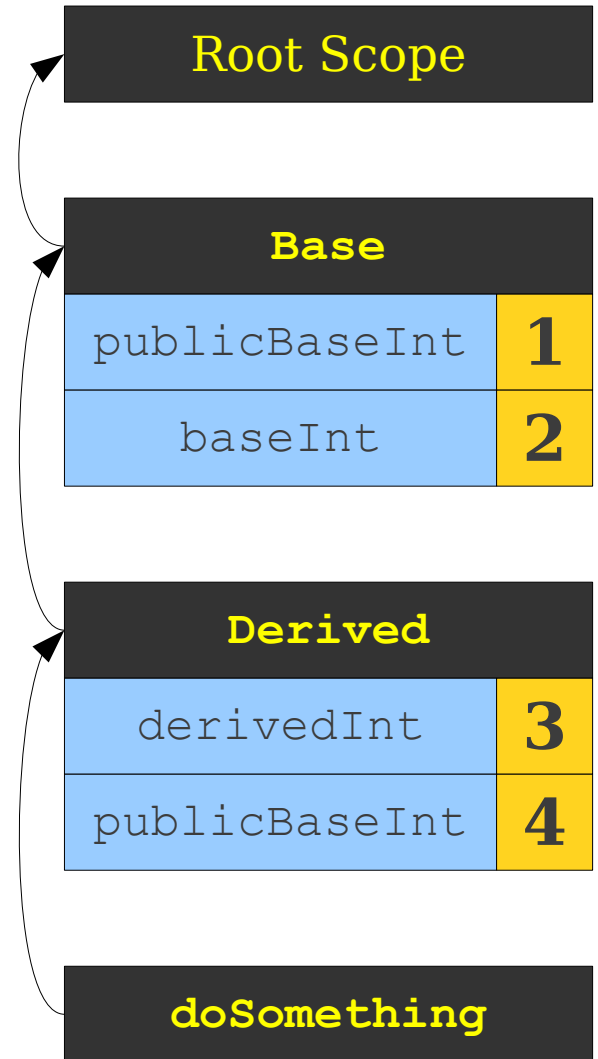
Scoping with Inheritance

```java
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
```

Root Scope

Base

publicBaseInt	1
baseInt	2

Derived

derivedInt	3
publicBaseInt	4

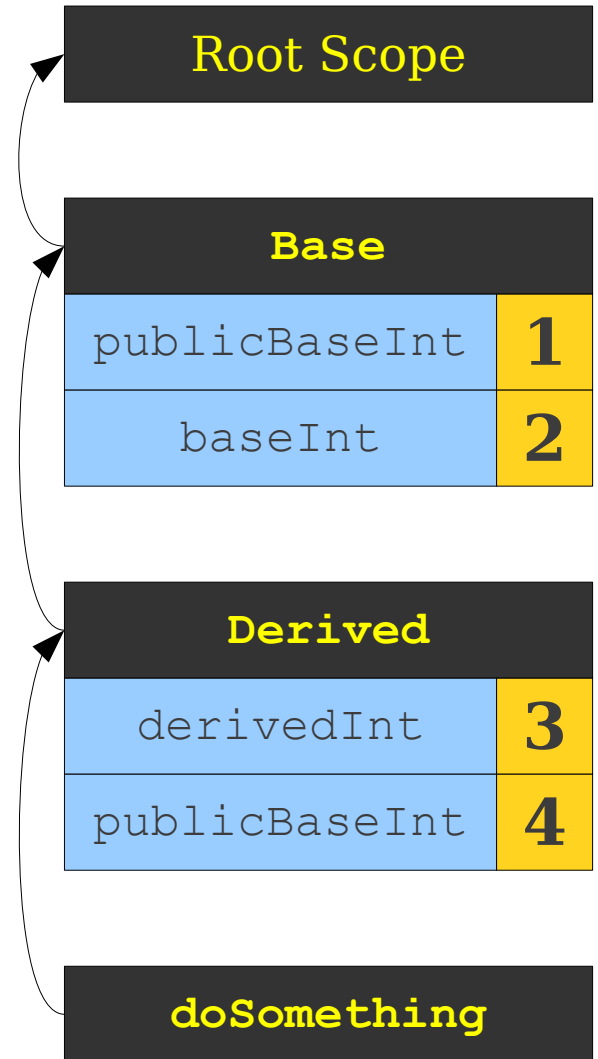**doSomething**

Scoping with Inheritance

```java
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
```


Root Scope

Base

| publicBaseInt | 1 |
| baseInt | 2 |

Derived

| derivedInt | 3 |
| publicBaseInt | 4 |

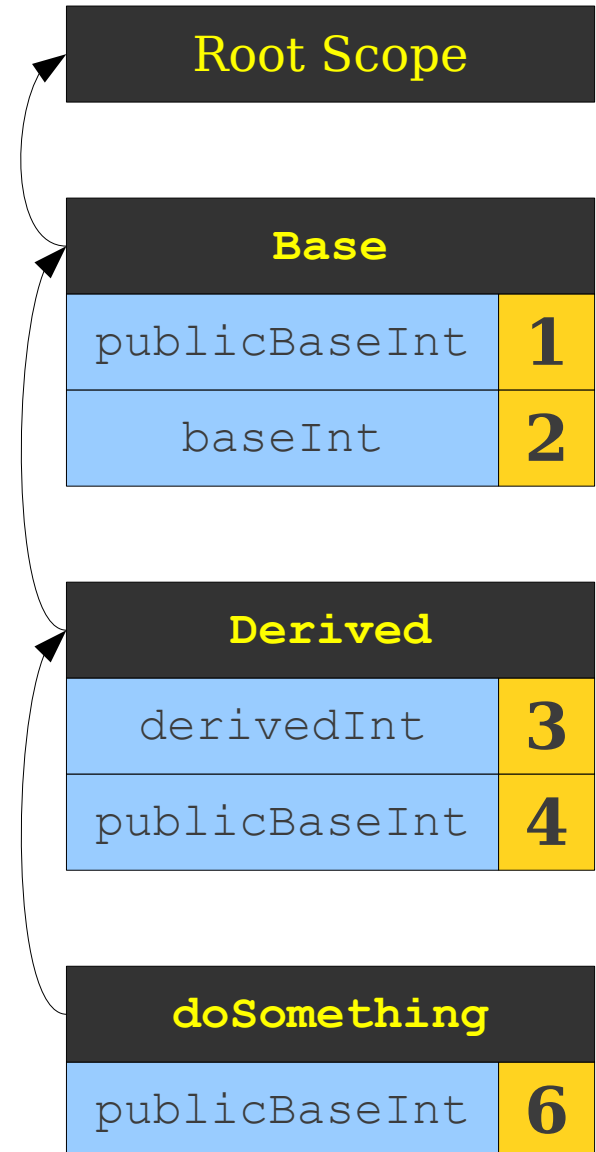**doSomething**

Scoping with Inheritance

```java
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
```

Root Scope	

Base	
publicBaseInt	1
baseInt	2

Derived	
derivedInt	3
publicBaseInt	4

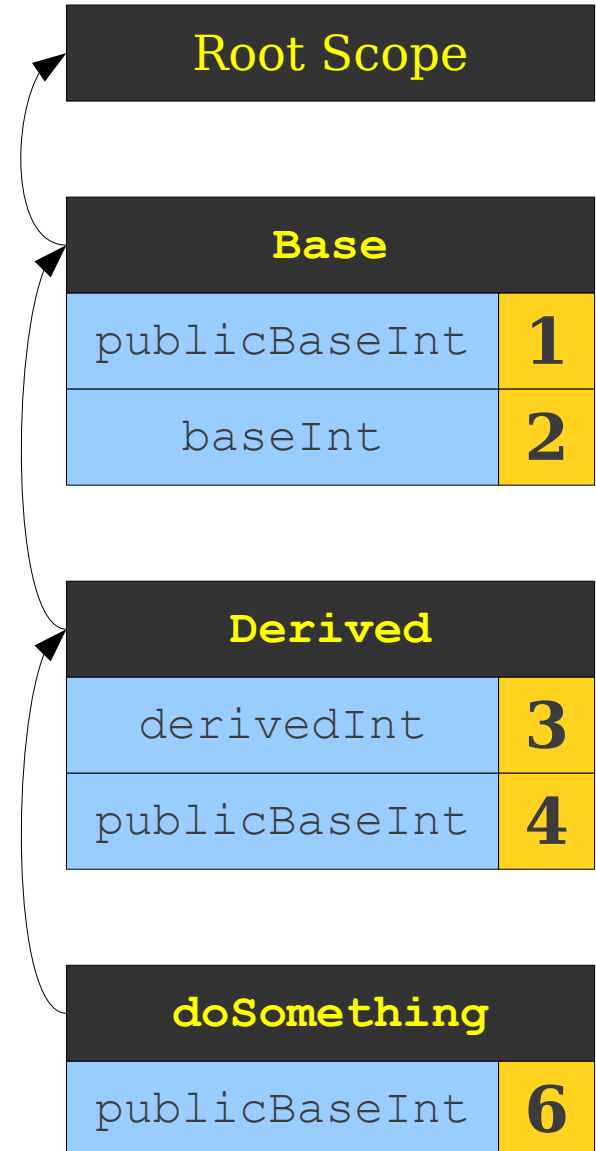| doSomething |
|---|

Scoping with Inheritance

```java
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
```

Root Scope

Base

publicBaseInt	1
baseInt	2

Derived

derivedInt	3
publicBaseInt	4

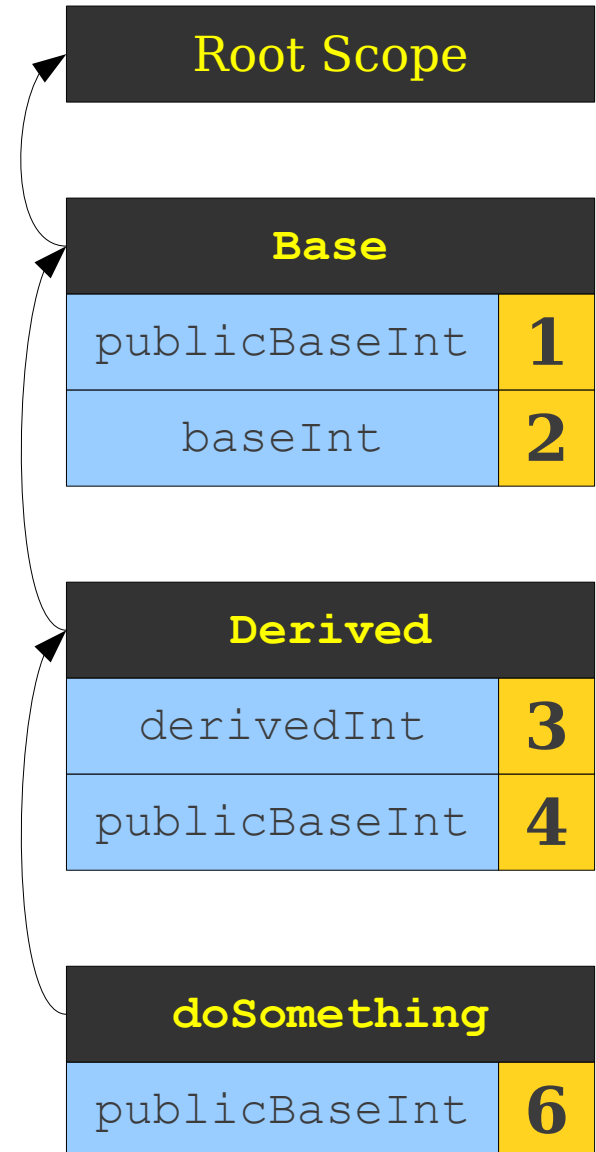**doSomething**

Scoping with Inheritance

```java
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
```

Root Scope

Base

publicBaseInt	1
baseInt	2

Derived

derivedInt	3
publicBaseInt	4

doSomething

publicBaseInt	6

Scoping with Inheritance

```java
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);
    }
}
```
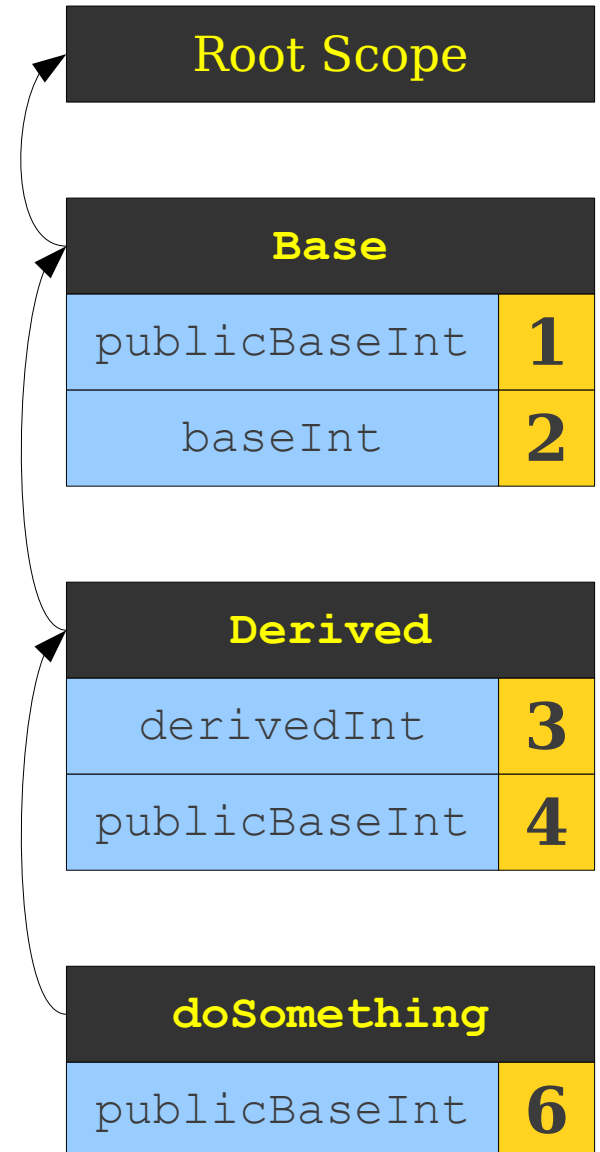
```
> 4
  2
  3
```

Root Scope

Base

publicBaseInt	1
baseInt	2

Derived

derivedInt	3
publicBaseInt	4

doSomething

publicBaseInt	6

Scoping with Inheritance

```java
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
```

Root Scope	

Base	
publicBaseInt	1
baseInt	2

Derived	
derivedInt	3
publicBaseInt	4

doSomething	
publicBaseInt	6

Scoping with Inheritance

```java
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
```

Scoping with Inheritance

```
public class Base {
    public int publicBaseInt = 1;
    protect...
}

public clas...
    public
    public

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
```

Typically the scope would also contain the names "**Base**" and "**Derived**," along with the function name "**doSomething**." For simplicity, I've left these out.

```
> 4
  2
  3
  6
```

Root Scope

Base	
publicBaseInt	**1**
baseInt	**2**

Derived	
derivedInt	**3**
publicBaseInt	**4**

doSomething	
publicBaseInt	**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.
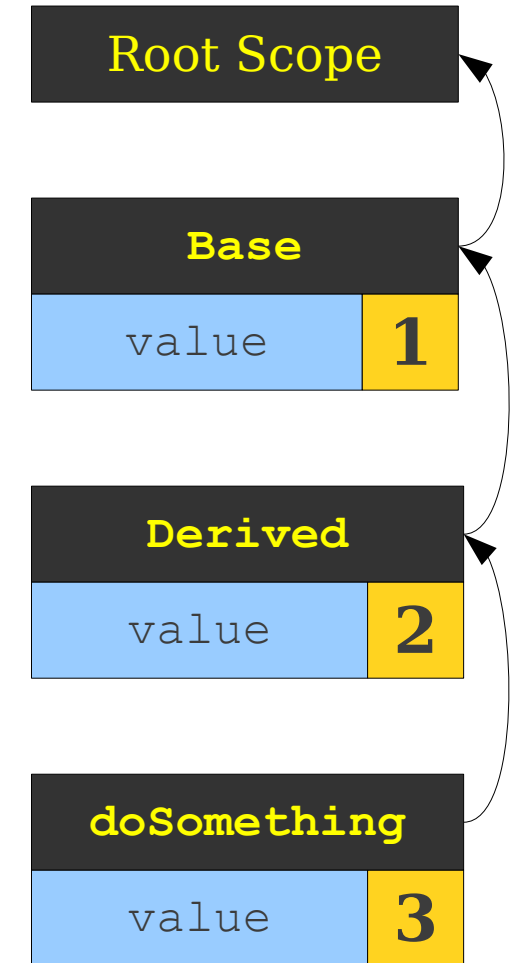
Explicit Disambiguation
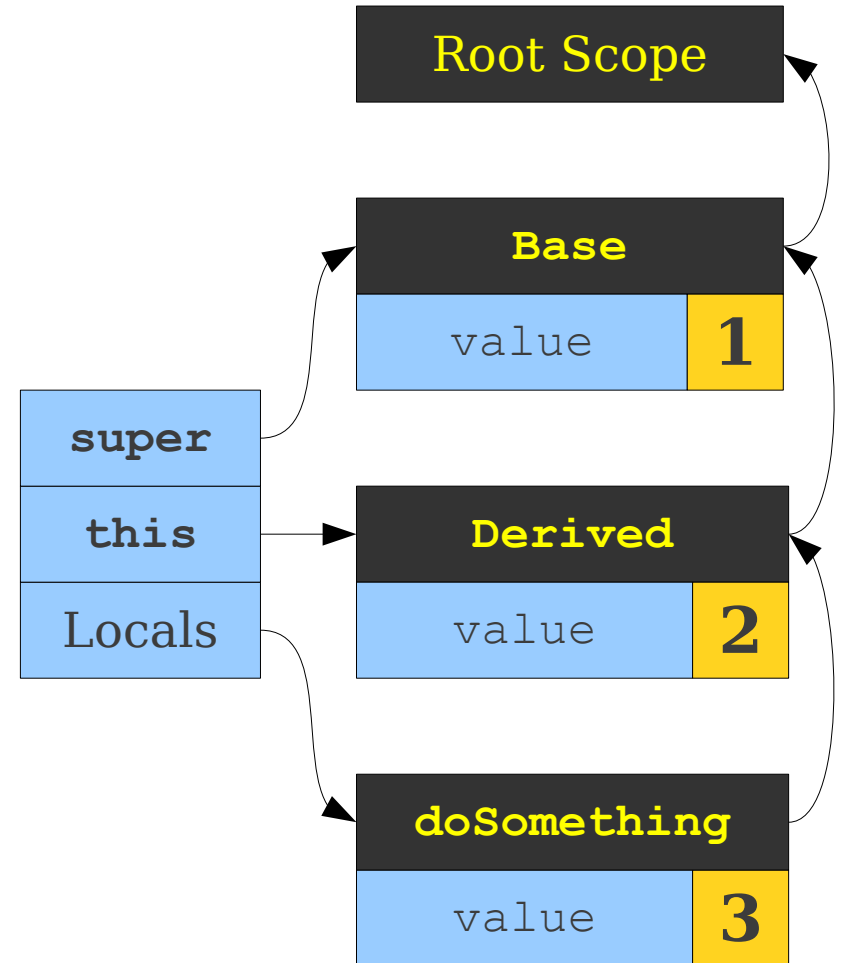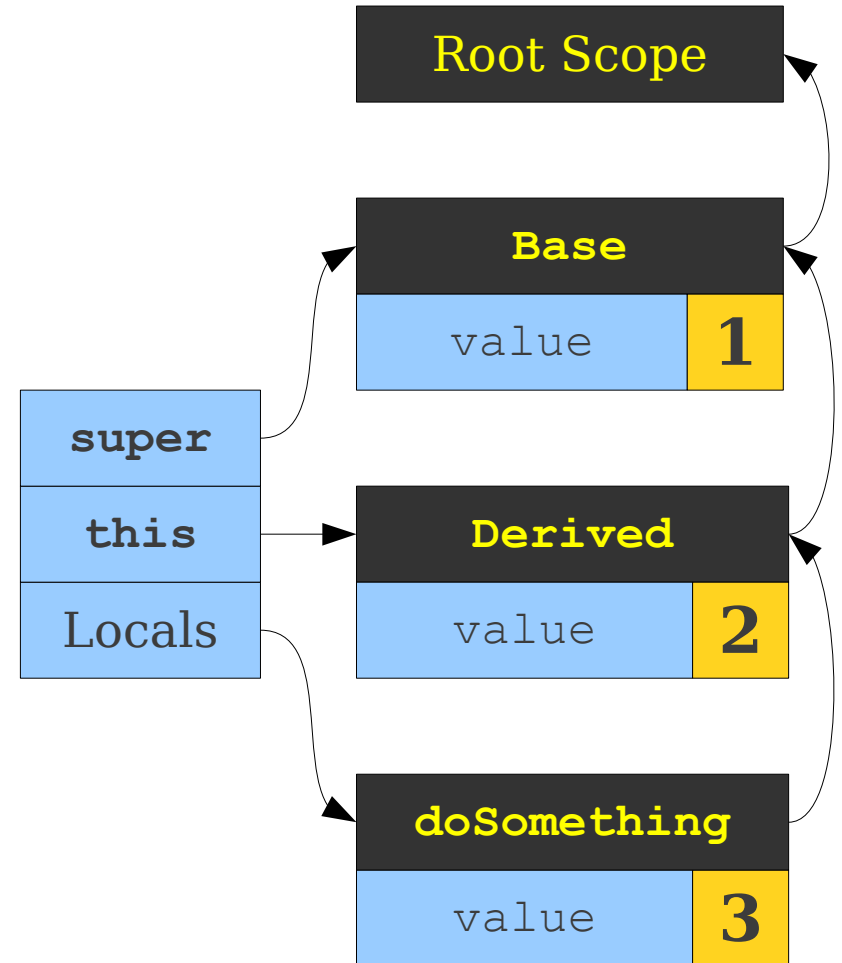
```java
public class Base {
    public int value = 1;
}

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);
    }
}
```
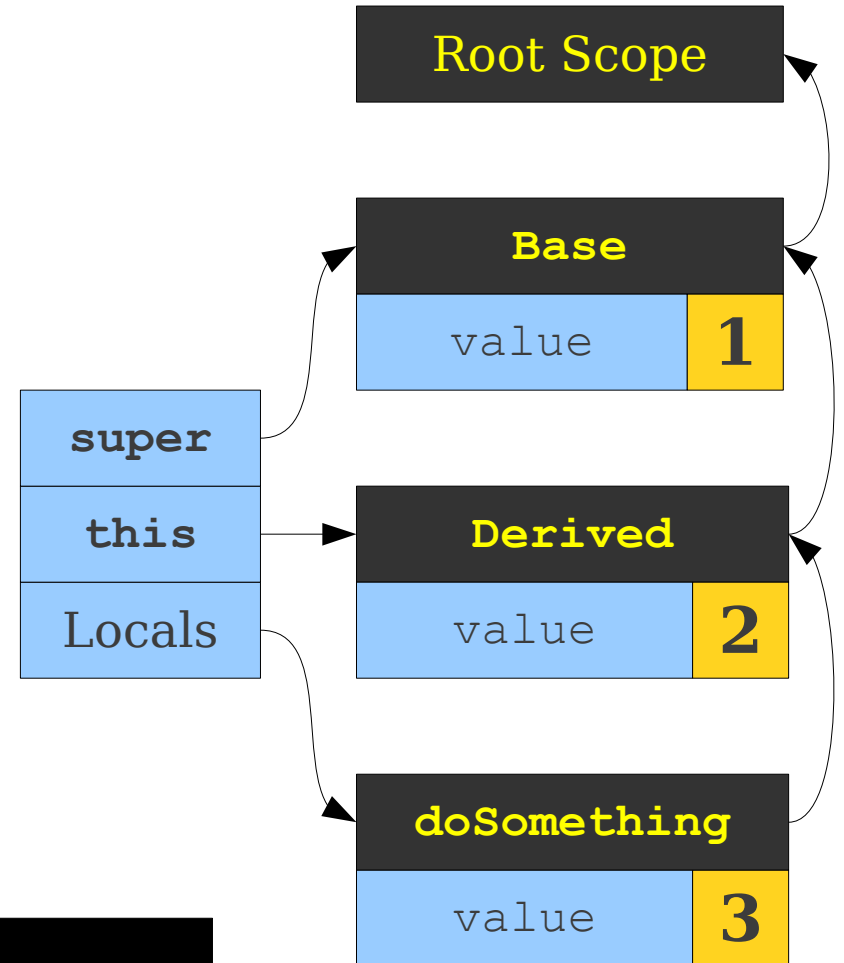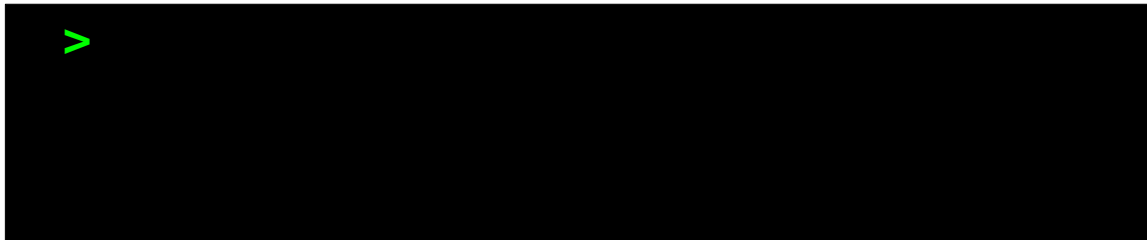
Explicit Disambiguation

```java
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);
    }
}
```
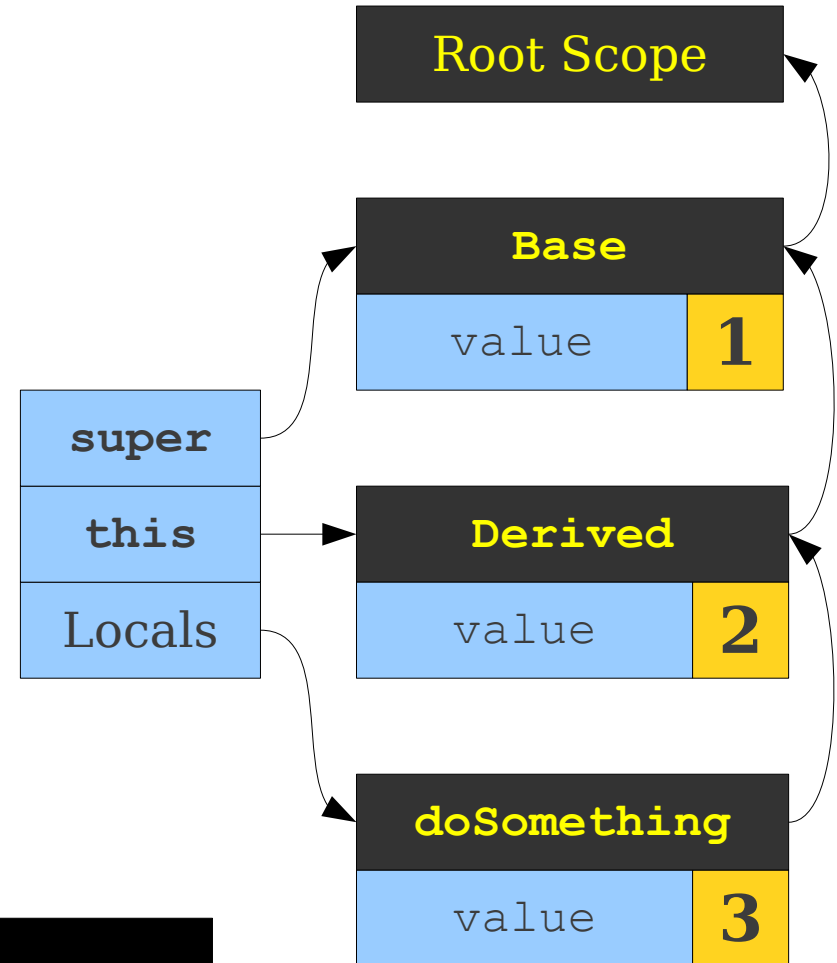
Explicit Disambiguation

```
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);
    }
}
```
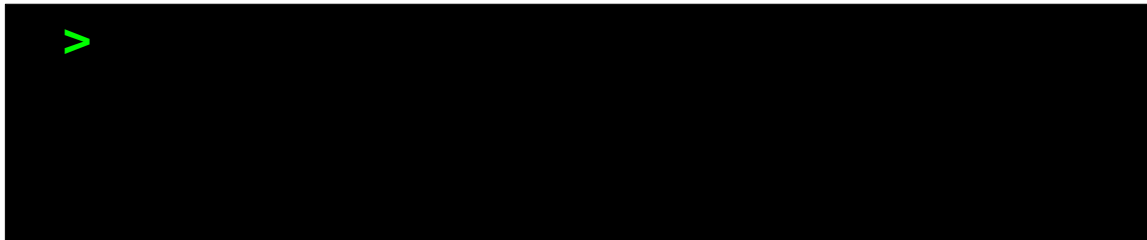
Explicit Disambiguation

```
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);
    }
}
```

Root Scope

Base

value | 1

super
this
Locals

Derived

value | 2

doSomething

value | 3

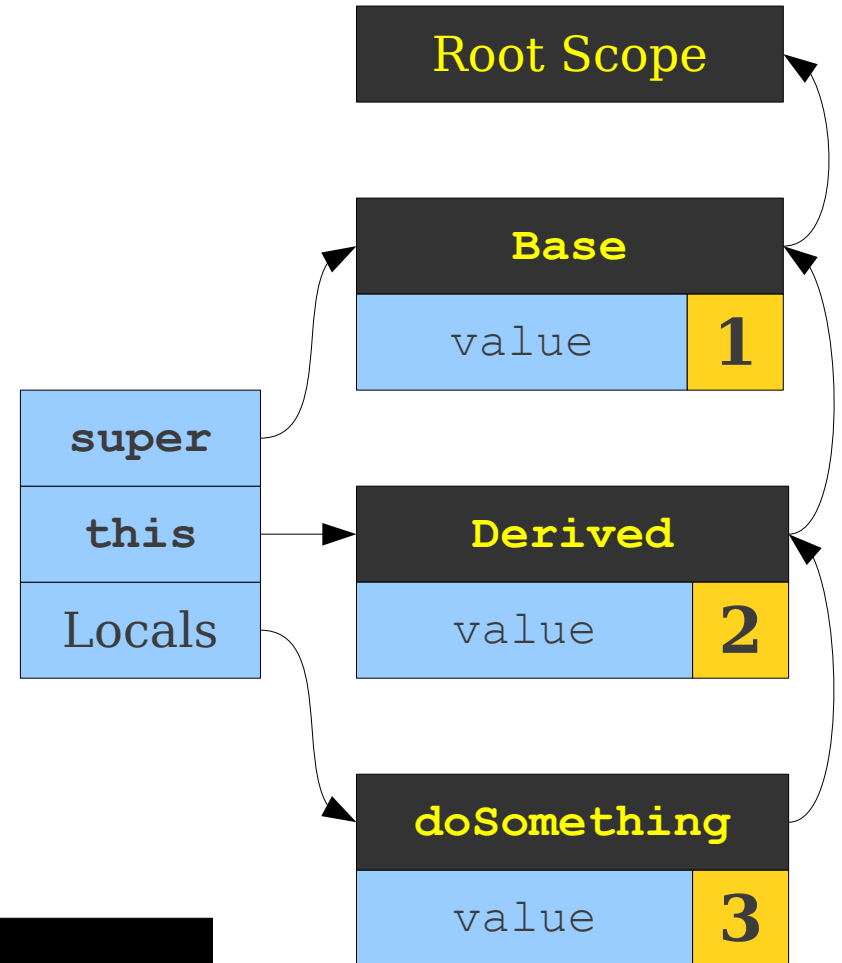# Explicit Disambiguation

```java
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);
    }
}
```

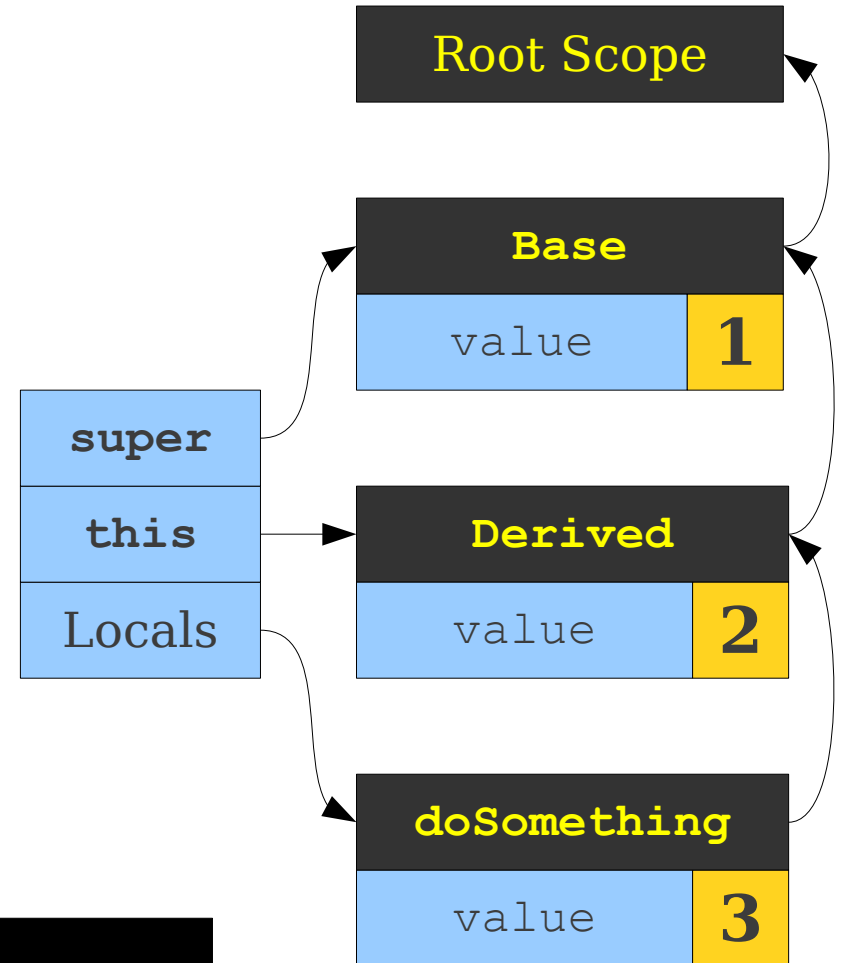# Explicit Disambiguation

```java
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

```java
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

```java
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);
    }
}
```
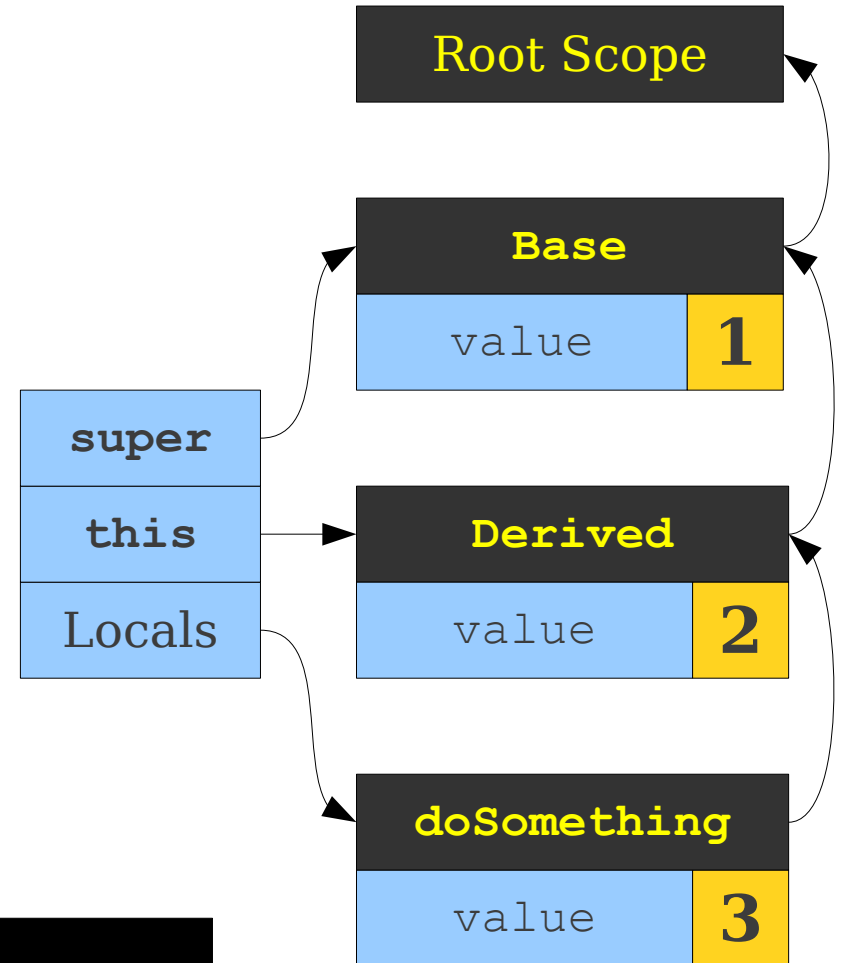


```
> 3
```

Explicit Disambiguation

```java
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);
    }
}
```
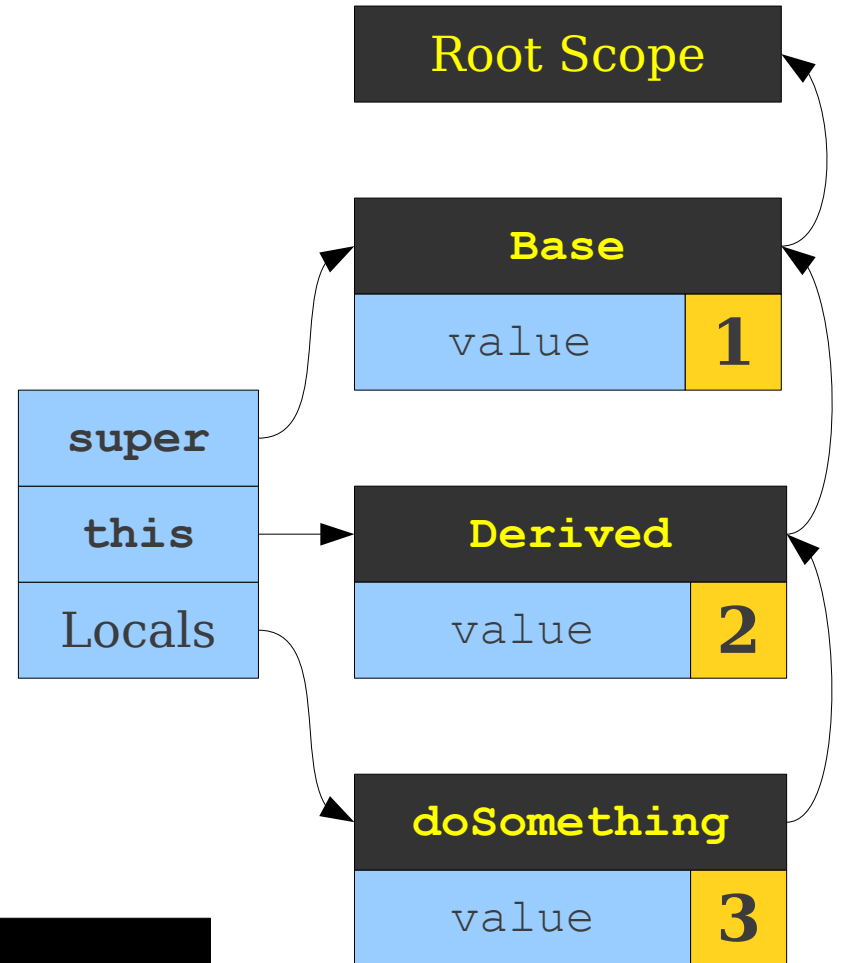


```
> 3
  2
```

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
```

Root Scope

Base
value 1

super

this

Locals

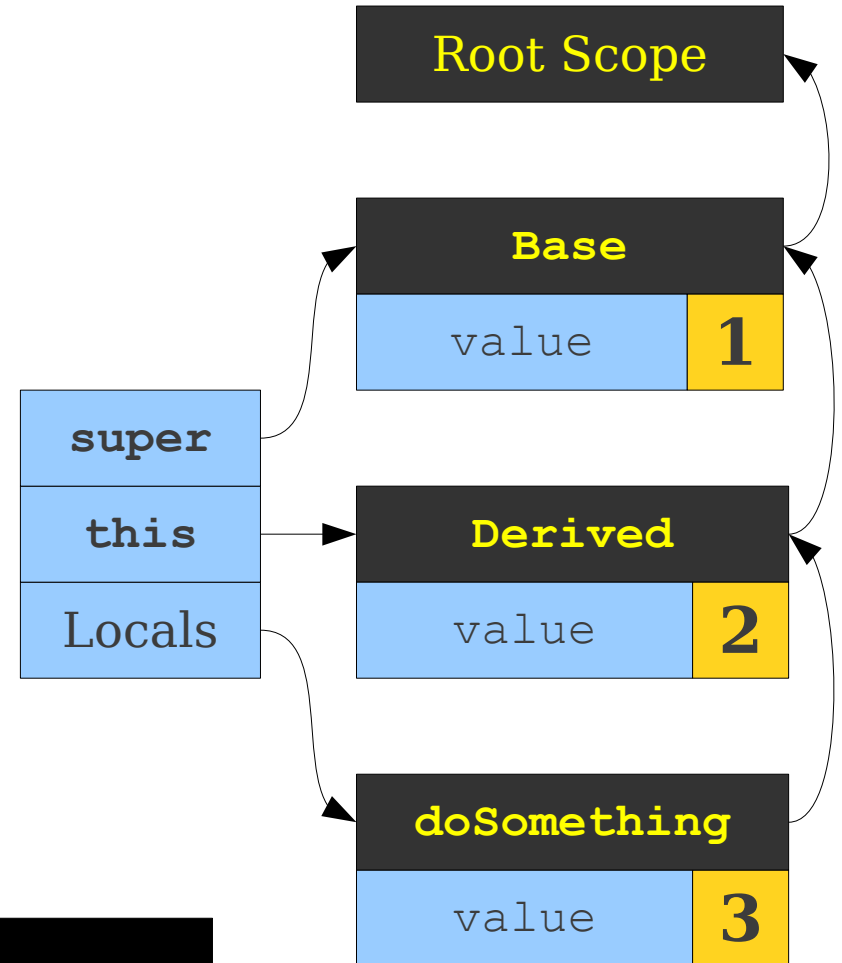Derived
value 2

doSomething
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
  1
```

Root Scope

Base
value 1

super
this
Locals

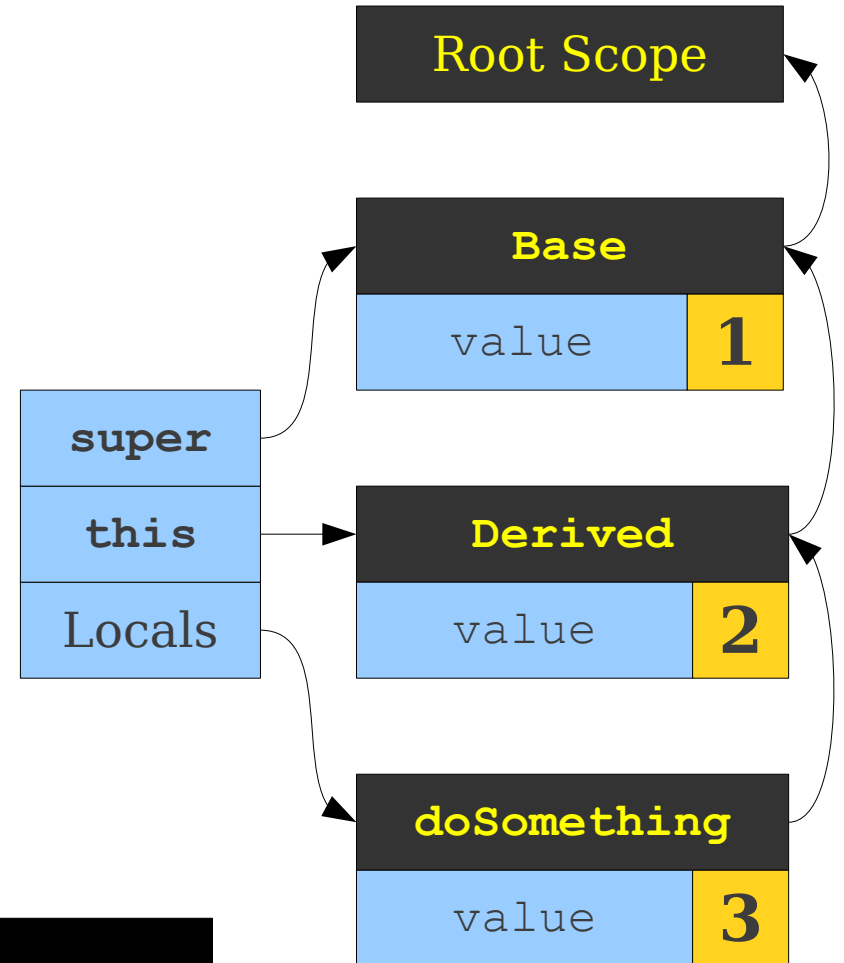Derived
value 2

doSomething
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
  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);
    }
}
```
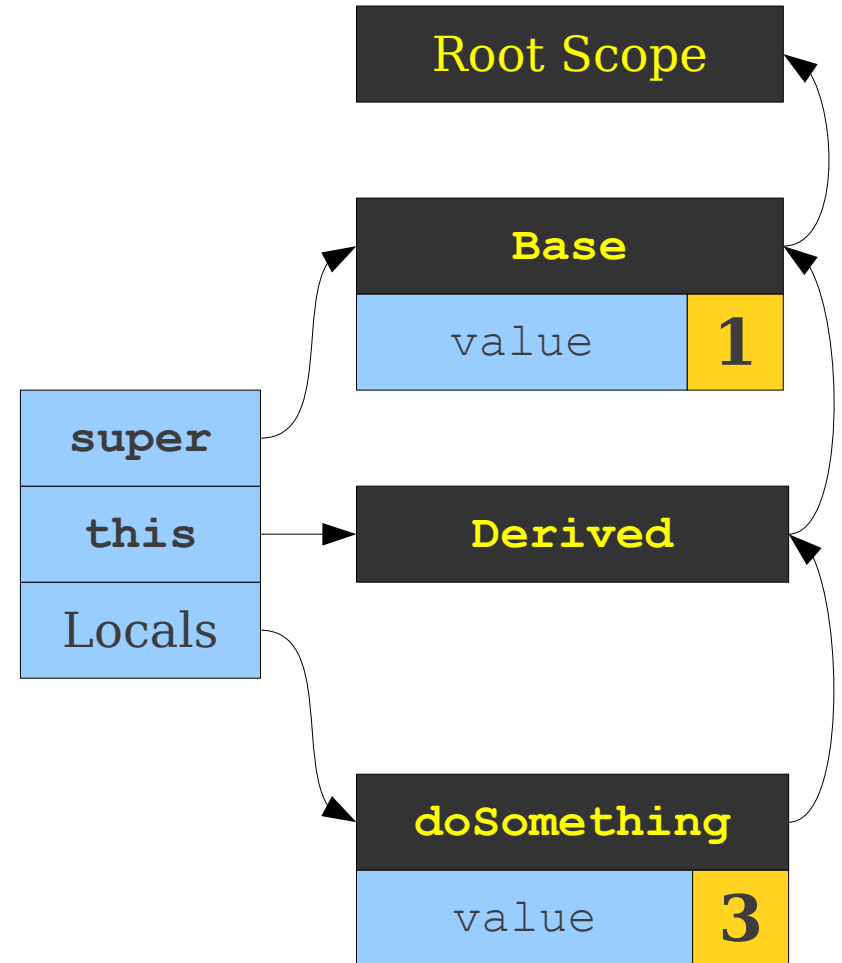
Root Scope

Base

value   1

super

this

Locals

Derived

doSomething

value   3

Explicit Disambiguation

```java
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);
    }
}
```
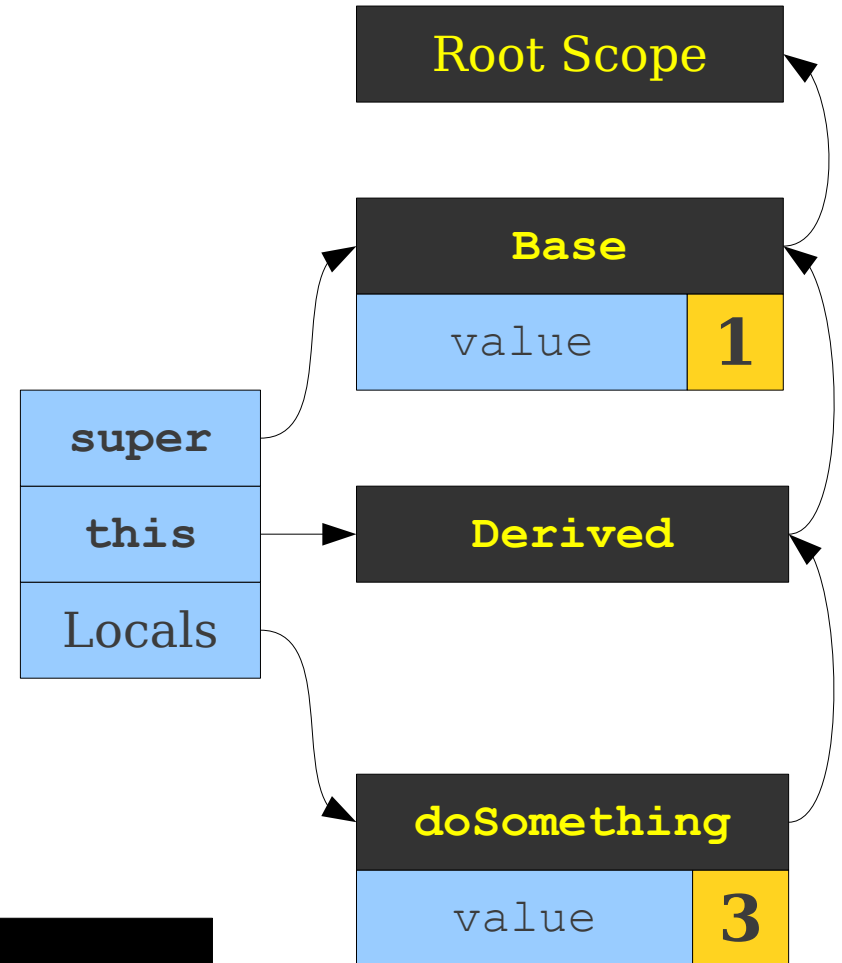
Explicit Disambiguation

```
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);
    }
}
```

Root Scope

Base

value | 1

super

this

Locals
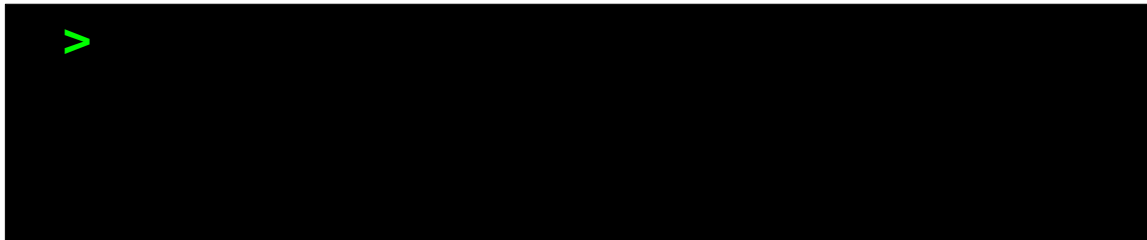
Derived

doSomething

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

Root Scope

Base

value | 1

super
this
Locals
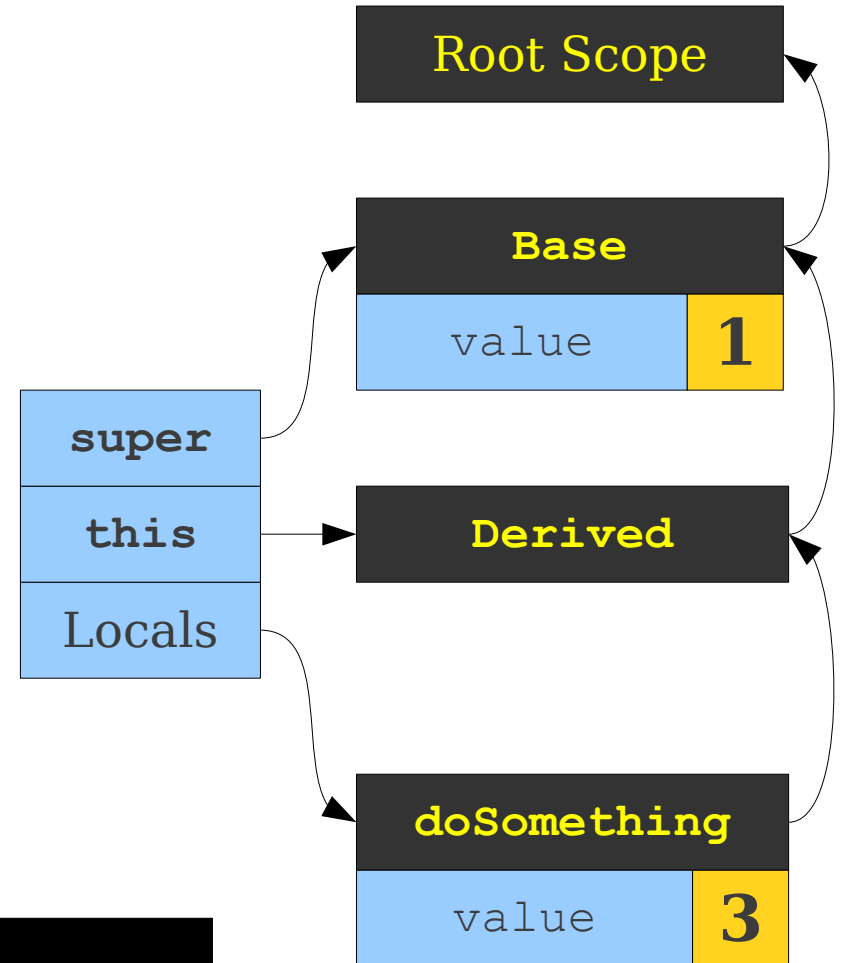
Derived

doSomething

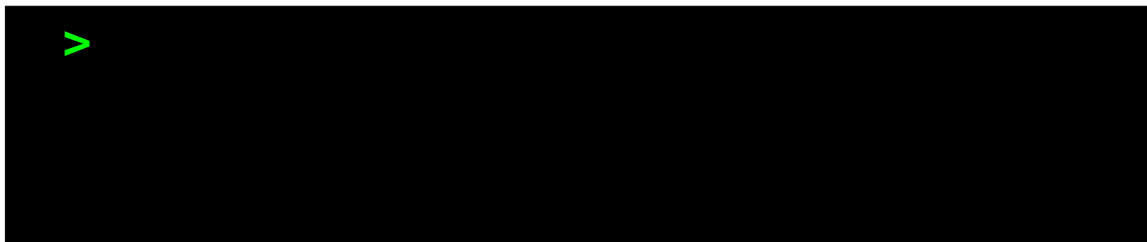value | 3

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);
    }
}
```
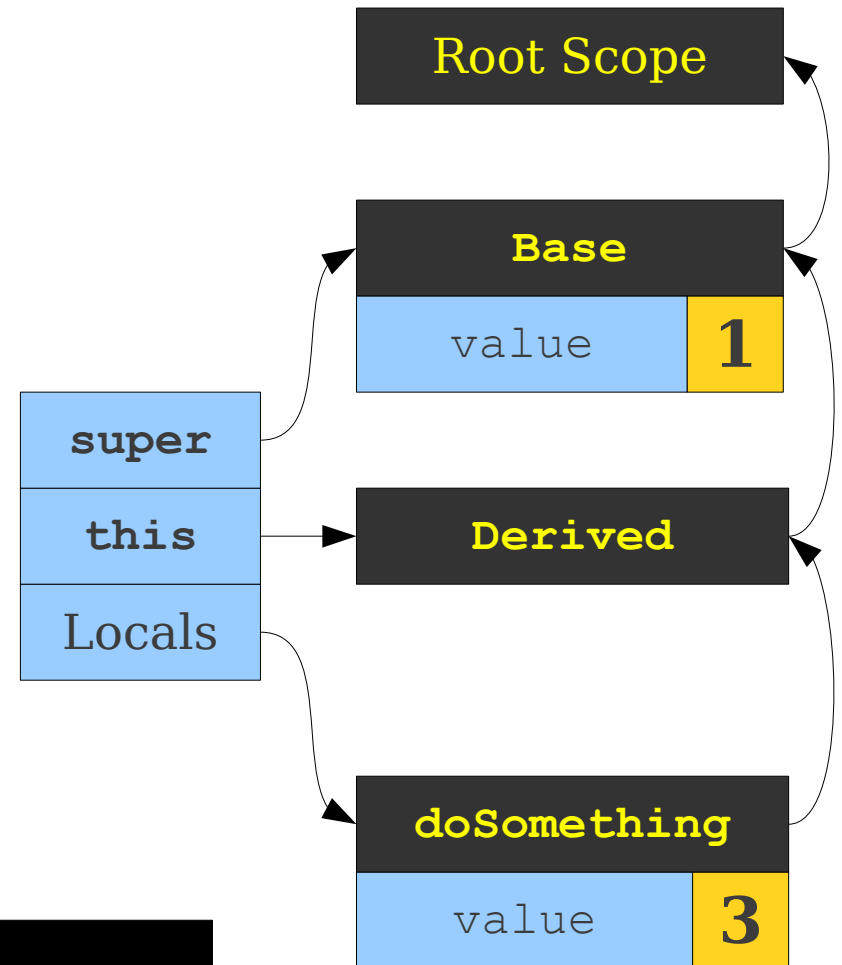
```
> 3
```

Root Scope

Base
value 1

super
this
Locals

Derived

doSomething
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
```

Root Scope

Base
value 1

super

this

Derived

Locals

doSomething
value 3
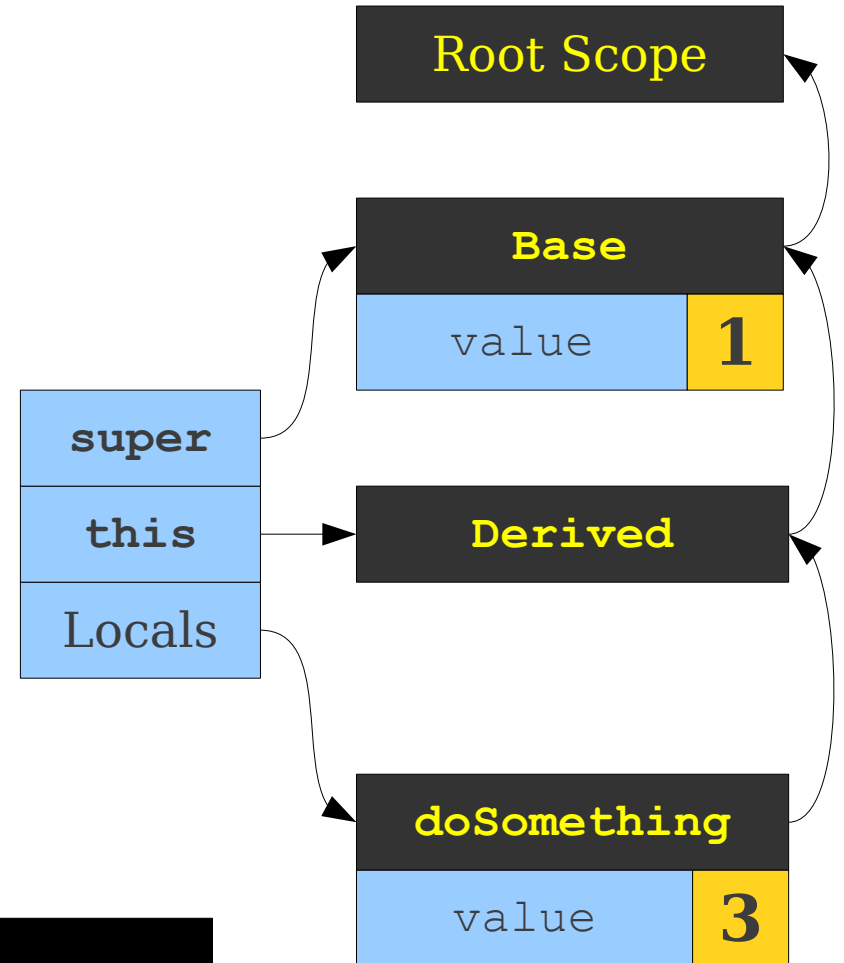
Explicit Disambiguation

```java
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
```
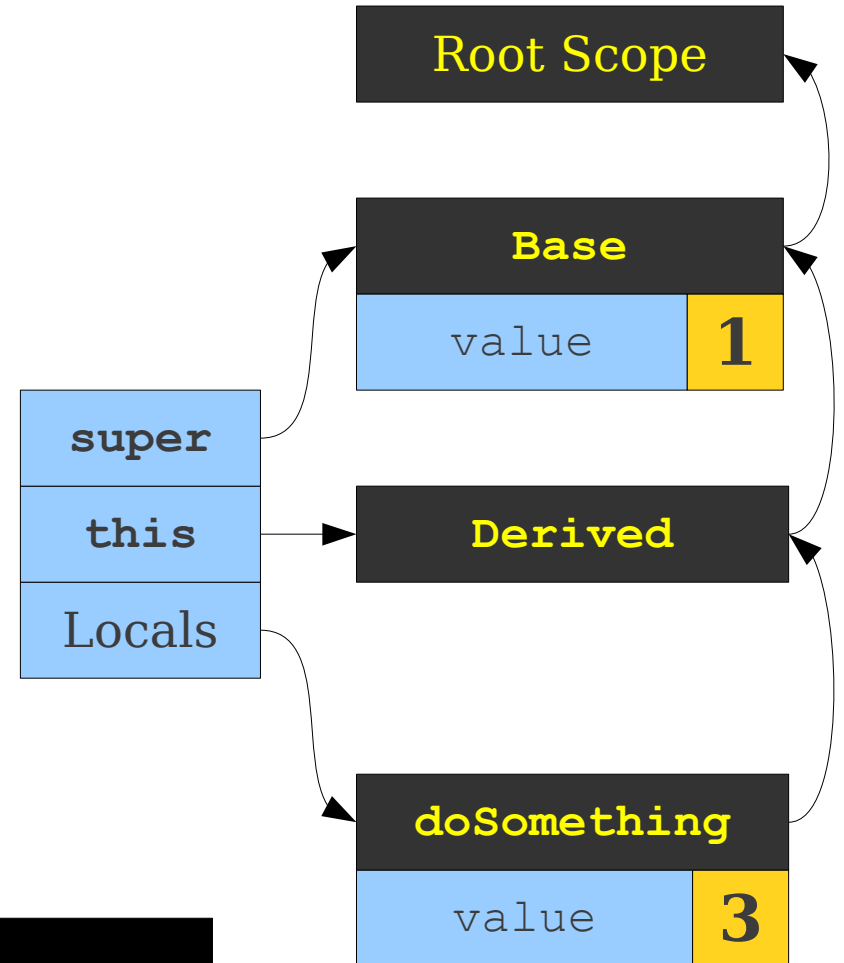
Explicit Disambiguation

```
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);
    }
}
```
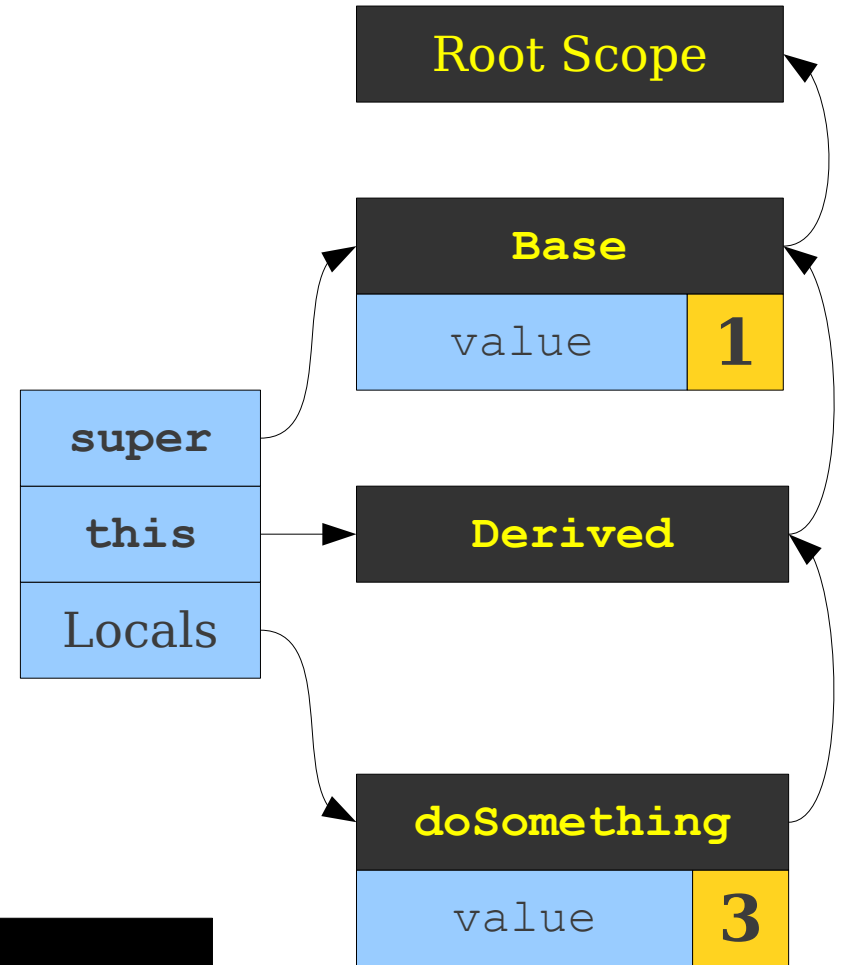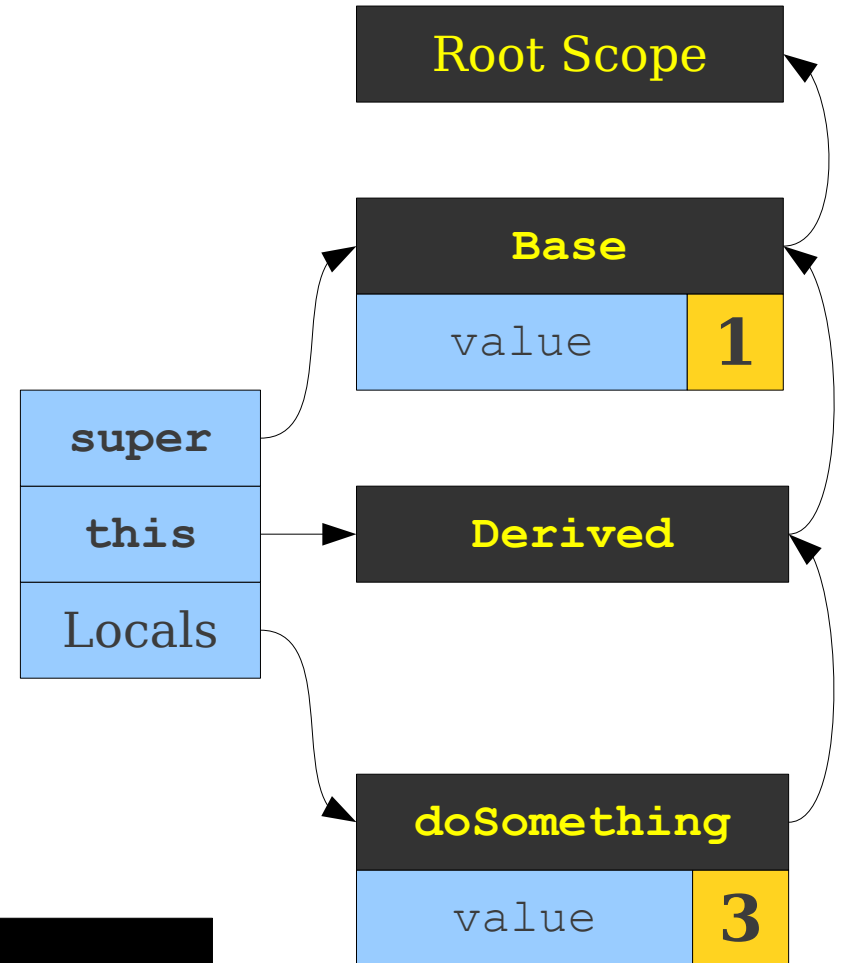
```
> 3
  1
  1
```

Explicit Disambiguation

```java
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);
    }
}
```
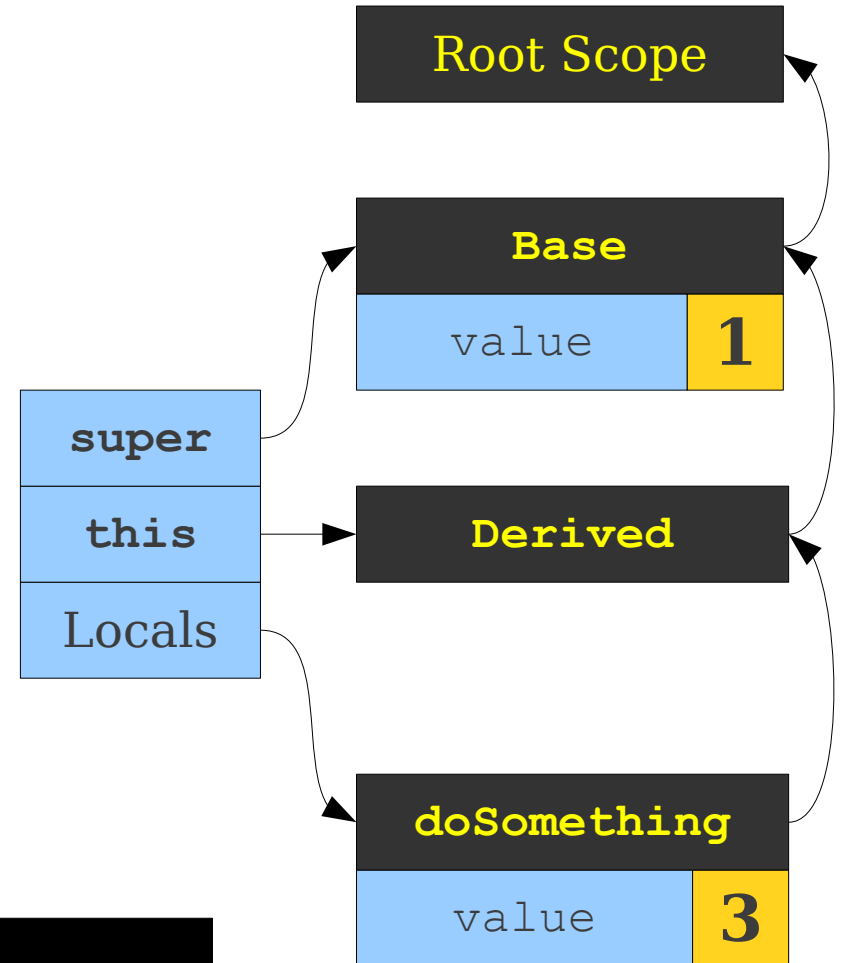
```
> 3
  1
  1
```

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 Practice

Scoping in C++ and Java

```cpp
class A {
public:
    /* … */

private:
    B* myB
};

class B {
public:
    /* … */

private:
    A* myA;
};
```

```java
class A {
    private B myB;
};

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

Scoping in C++ and Java

```
class A {
public:
    /* … */

private:
    B* myB
};
```

Error: B not
declared

```
class B {
public:
    /* … */

private:
    A* myA;
};
```

```
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.
  - **L**L(1), **L**R(0), LA**L**R(1), etc.
- Since we only need one token of lookahead, we can do scanning and parsing simultaneously in one pass over the file.
- Some compilers can combine scanning, parsing, semantic analysis, and code generation into the same pass.
  - These are called **single-pass compilers**.
- Other compilers rescan the input multiple times.
  - These are called **multi-pass compilers**.

Single- and Multi-Pass Compilers

- Some languages are designed to support single-pass compilers.

  - e.g. C, C++.

- Some languages *require* multiple passes.

  - e.g. Java, **Decaf**.

- Most modern compilers use a huge number of passes over the input.

Scoping in Multi-Pass Compilers

- Completely parse the input file 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, though they are logically distinct.

Scoping with Multiple Inheritance

```cpp
class A {
public:
    int x;
};


class B {



};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

Scoping with Multiple Inheritance

```
class A {
public:
    int x;
};

class B {



};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

Scoping with Multiple Inheritance

Root Scope

A

x

```
class A {
public:
    int x;
};

class B {



};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

Scoping with Multiple Inheritance

```
class A {
public:
    int x;
};

class B {



};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

Scoping with Multiple Inheritance

```cpp
class A {
public:
    int x;
};

class B {



};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

Scoping with Multiple Inheritance

```
class A {
public:
    int x;
};


class B {



};


class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

Root Scope

A

x

B

C

Scoping with Multiple Inheritance

```
class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

Scoping with Multiple Inheritance

```
class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```
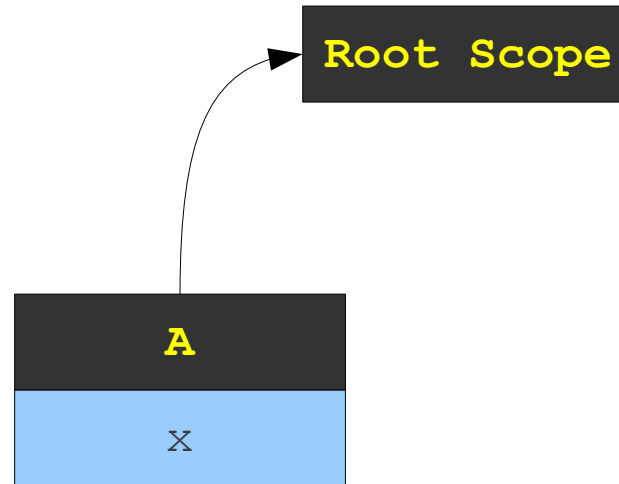
Scoping with Multiple Inheritance

```cpp
class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

Scoping with Multiple Inheritance

```
class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

Root Scope

A

x

B

x

C

Ambiguous –
which x?

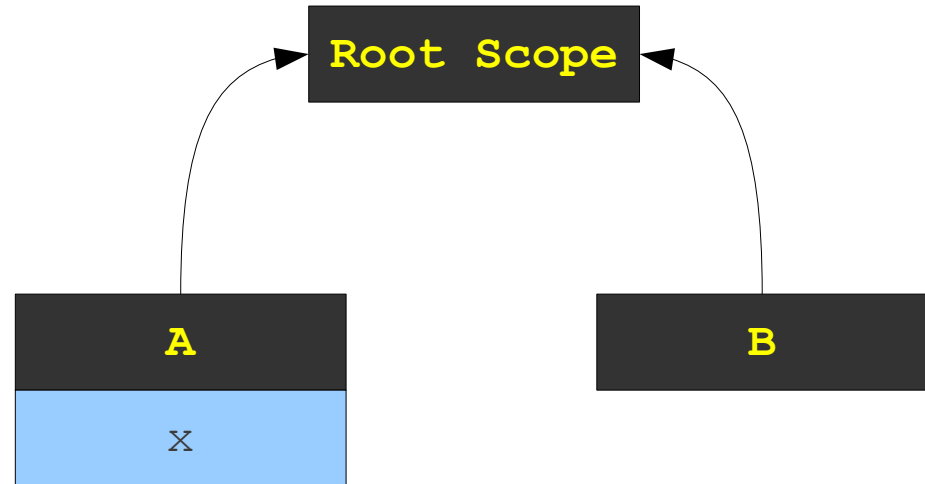# Scoping with Multiple Inheritance

```
class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << A::x << endl;
    }
}
```

Scoping with Multiple Inheritance

```
class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

Scoping with Multiple Inheritance

```
class A {
public:
    int x;
};

class B {



};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```
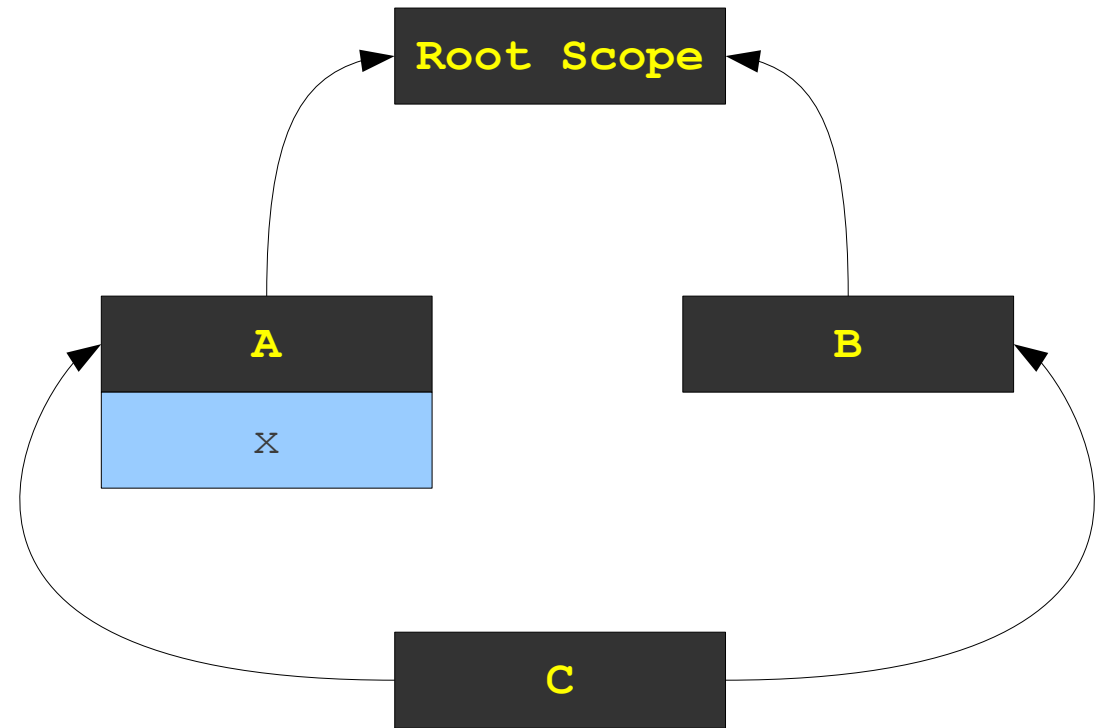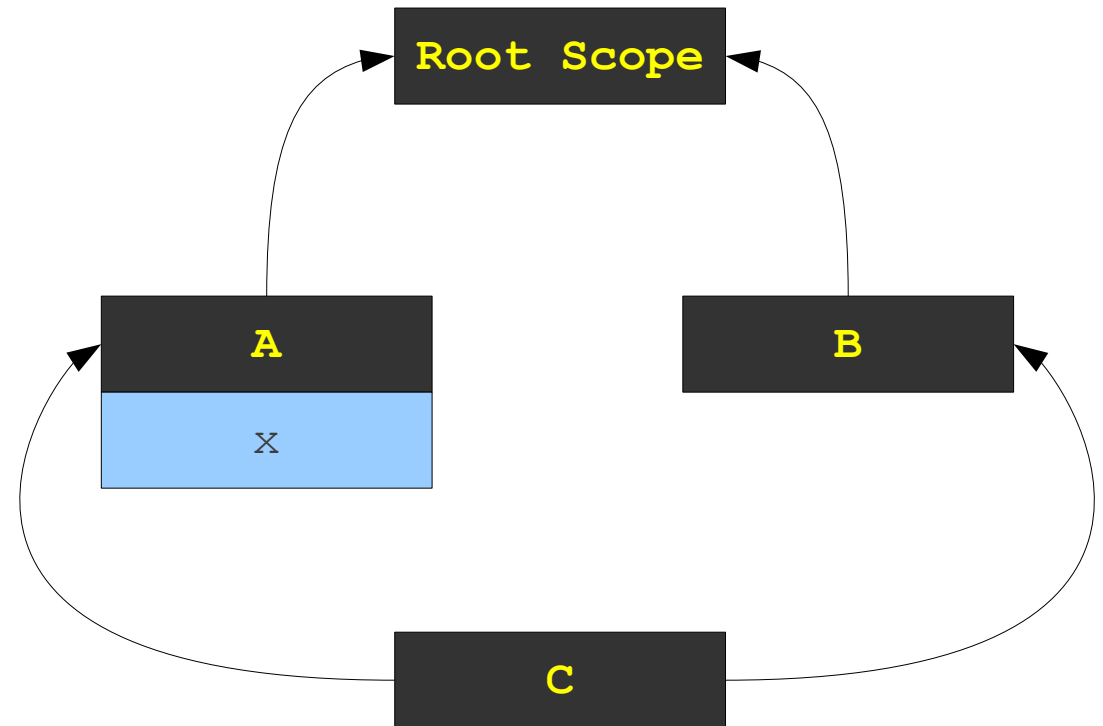
Scoping with Multiple Inheritance

```
class A {
public:
    int x;
};

class B {



};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

Scoping with Multiple Inheritance

```cpp
int x;

class A {
public:
    int x;
};

class B {



};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```
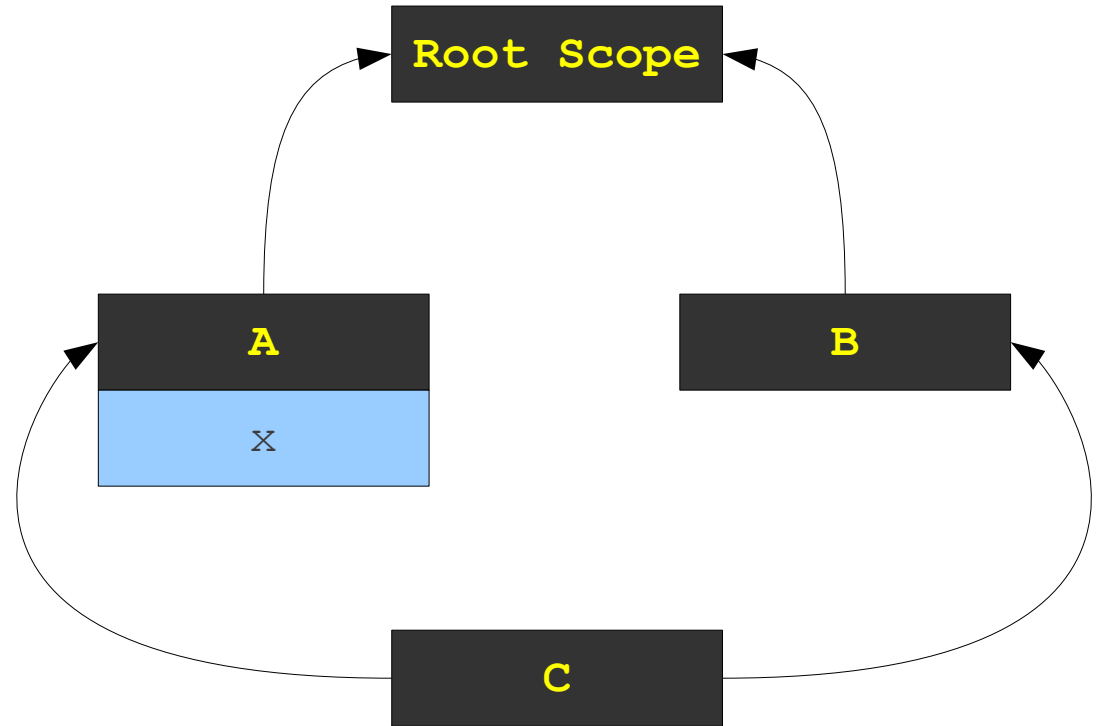
Scoping with Multiple Inheritance

```
int x;

class A {
public:
    int x;
};

class B {



};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

Scoping with Multiple Inheritance

```
int x;

class A {
public:
    int x;
};

class B {



};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```
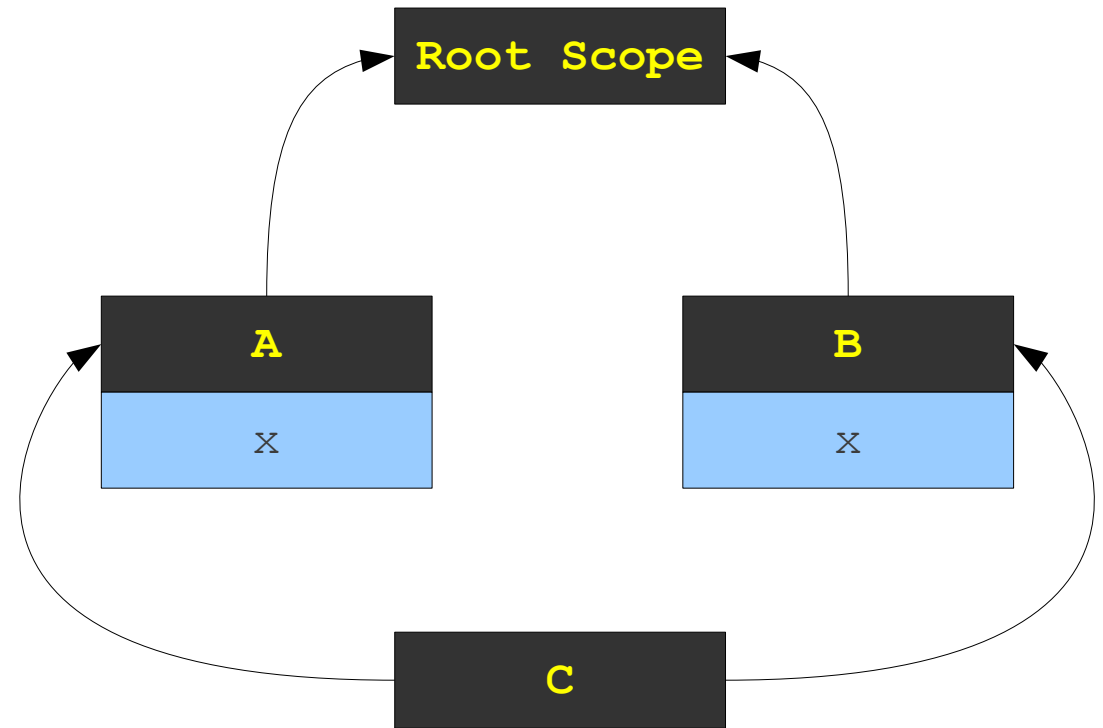
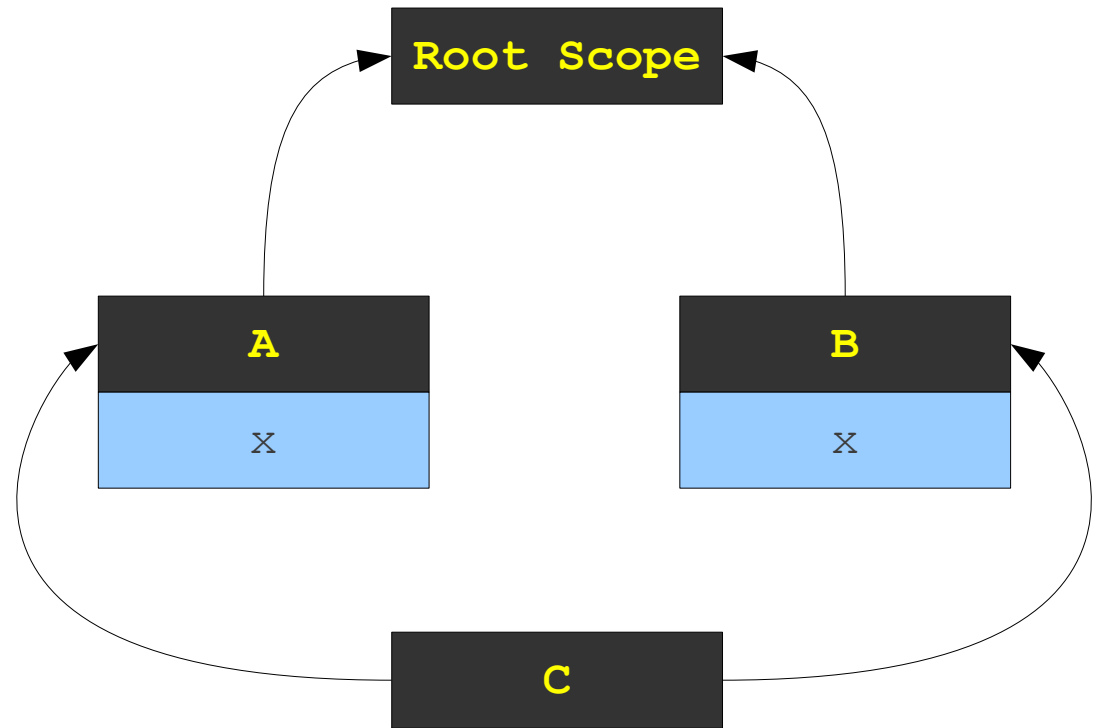**Root Scope**

x

A

x

B

C

(Simplified) C++ Scoping Rules

- Inside of a class, search the entire class hierarchy to see the set of names that can be found.

  - This uses the standard scoping lookup.

- If only one name is found, the lookup succeeds unambiguously.

- If more than one name is found, the lookup is ambiguous and requires disambiguation.

- Otherwise, restart the search from outside the class.

Dynamic Scoping

Static and Dynamic Scoping

- The scoping we've seen so far is called **static scoping** and is done at compile-time.

  - Names refer to lexically related variables.

- Some languages use **dynamic scoping**, which is done at runtime.

  - Names refer to the variable with that name that is most 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();
```

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**

>

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**

```
> 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
>
```

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**
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**

```
> 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**

```
> 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
>
```

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**
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**
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

```
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, Common LISP.

- Often implemented by preserving symbol table at runtime.

- Often less efficient than static scoping.

  - Compiler cannot "hardcode" locations 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.

Next Time

- **Type Checking**
  - Types as a proof system.
  - Static and dynamic types.
  - Types as a partial order.