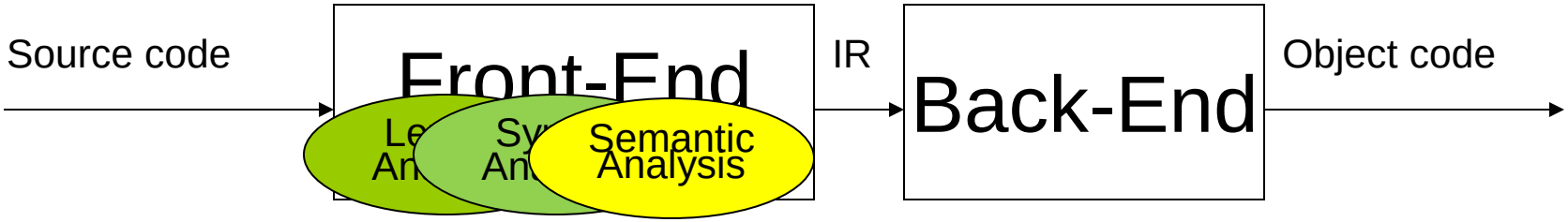
Compiler Design

Lecture 7:
Semantic Analysis

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book

Semantic Analysis



- Semantic Analysis:
  - Beyond lexical and Syntactic analysis

Outline

- **Introduction**
- Scope-Checking
- Type-Checking

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

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

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

Validity versus Correctness

Incorrect, should be "return n;"

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

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

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

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.

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.

Tasks

- Scope-Checking
  - How can we tell what object a particular identifier refers to?
  - How do we store this information?

- Type-Checking
  - How can we tell whether expressions have valid types?
  - How do we know all function calls have valid arguments?

Outline

- Introduction
- **Scope-Checking**
  - Scoping in Object-Oriented Languages
  - Single- and Multi-Pass Compilers
  - Dynamic and Static Scoping
- Type-Checking

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:

```
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:

```
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

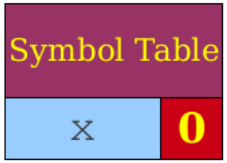
Symbol Tables: The Intuition

```
 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

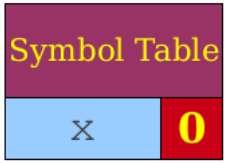
Symbol Tables: The Intuition

```
 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 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	
x	0
z	1

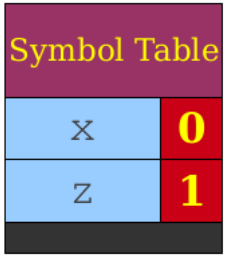
Symbol Tables: The Intuition

```
 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 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: }
```

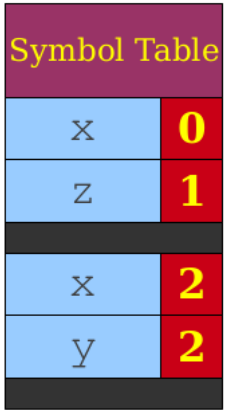
Symbol Table	
x	0
z	1
x	2
y	2

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 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 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 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 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 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

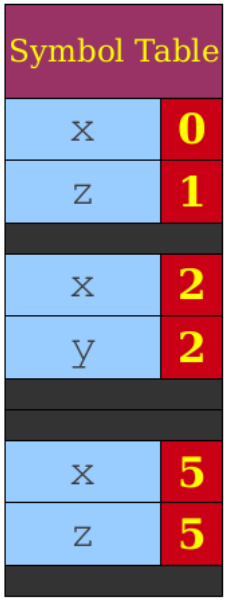
Symbol Tables: The Intuition

```
 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

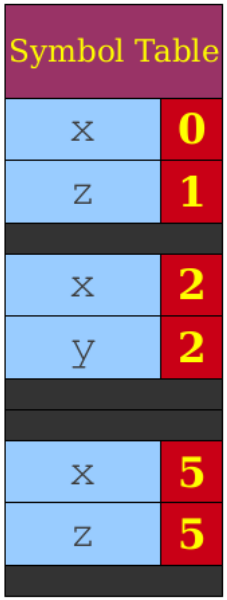
Symbol Tables: The Intuition

```
 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 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 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 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 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 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.

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

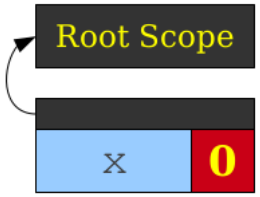
```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```

Another View of Symbol Tables

Root Scope

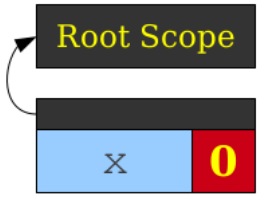
```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```

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: }
```

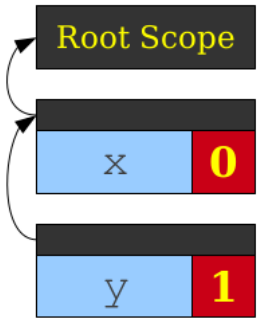
Another View of Symbol Tables




```
 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: }
```

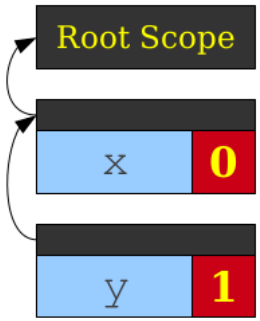
Another View of Symbol Tables

```
 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: }
```



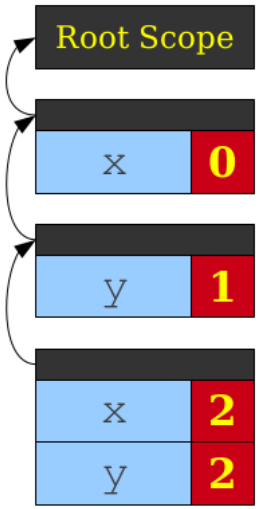
Another View of Symbol Tables

```
 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: }
```



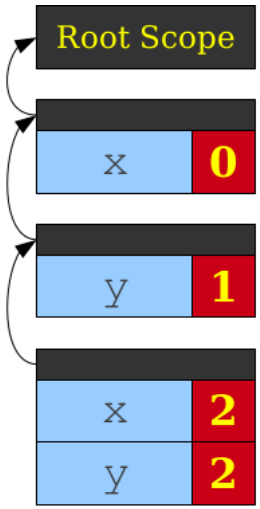
Another View of Symbol Tables

```
 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: }
```



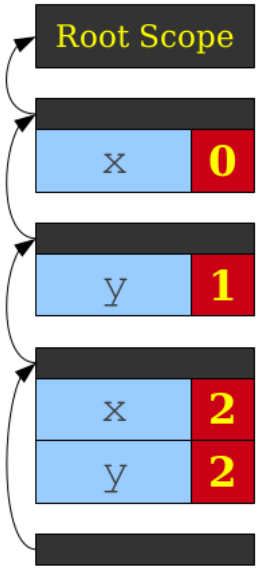
Another View of Symbol Tables

```
 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: }
```



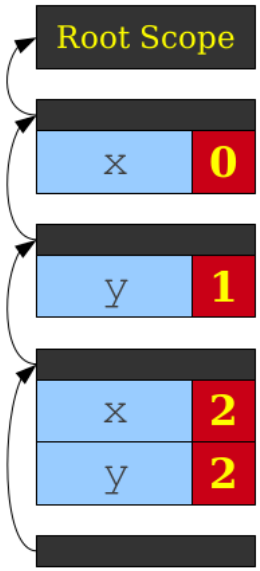
Another View of Symbol Tables

```
 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: }
```



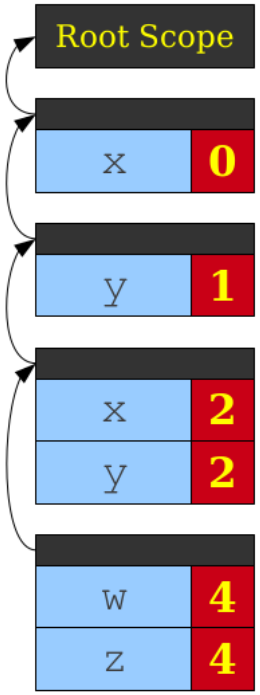
Another View of Symbol Tables

```
 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: }
```



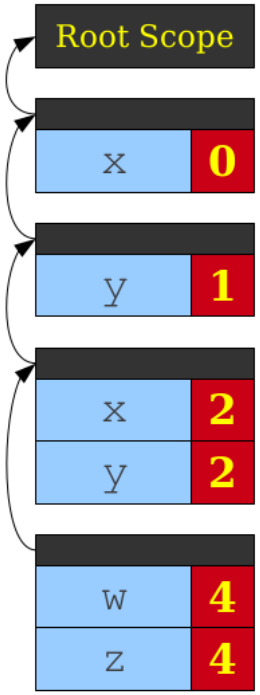
Another View of Symbol Tables

```
 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: }
```



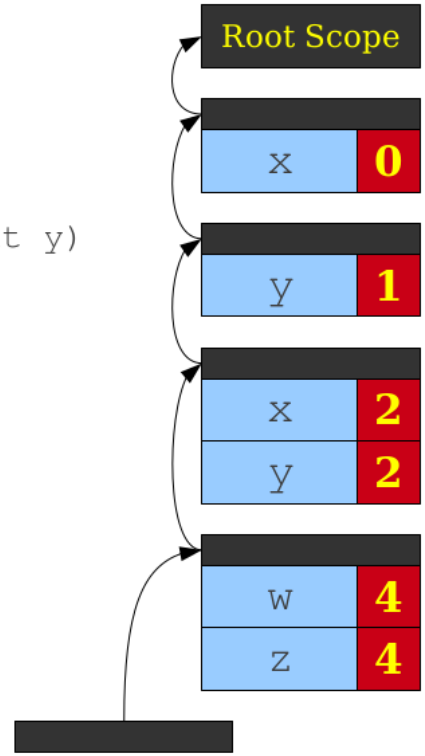
Another View of Symbol Tables

```
 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: }
```



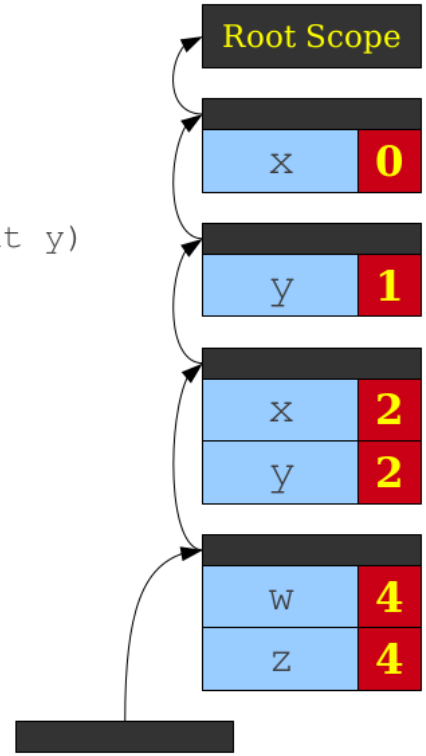
Another View of Symbol Tables

```
 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: }
```



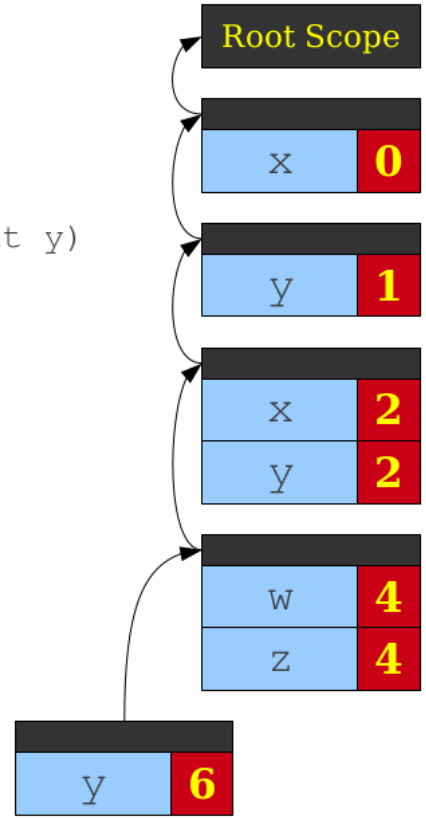
Another View of Symbol Tables

```
 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: }
```



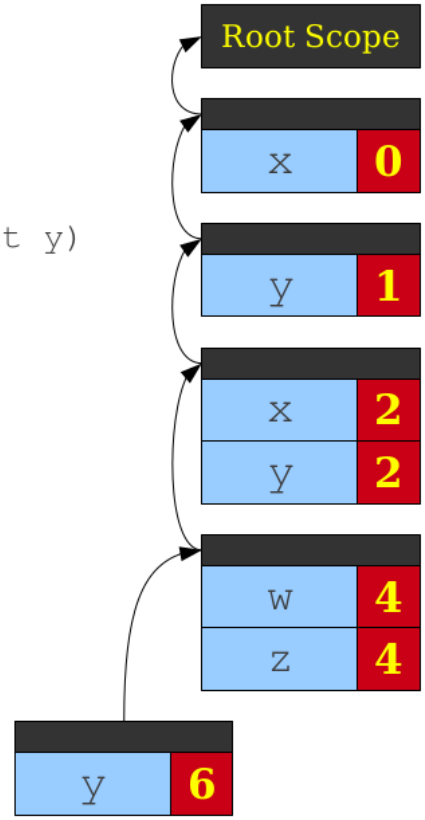
Another View of Symbol Tables

```
 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: }
```



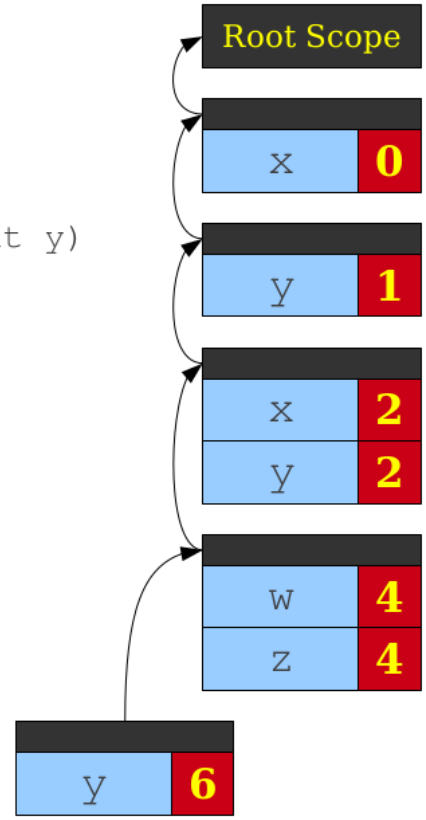
Another View of Symbol Tables

```
 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: }
```



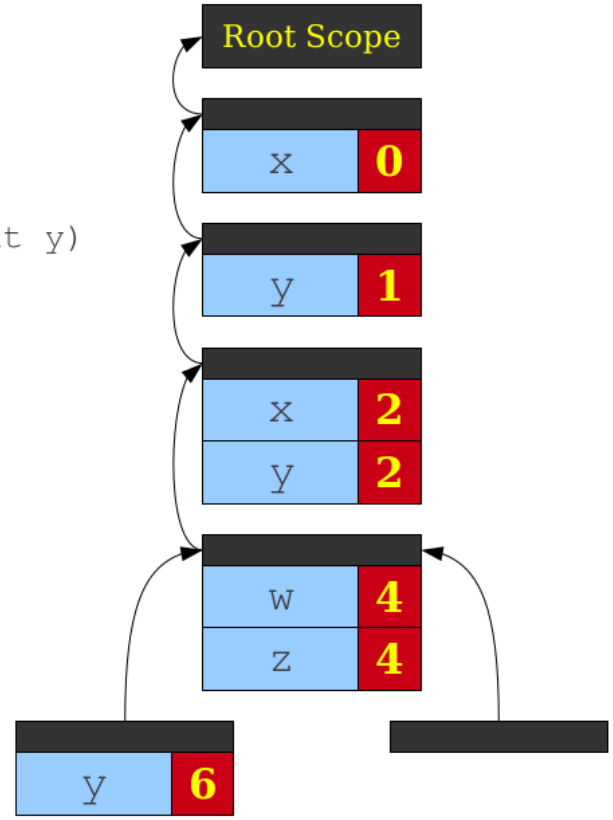
Another View of Symbol Tables

```
 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: }
```



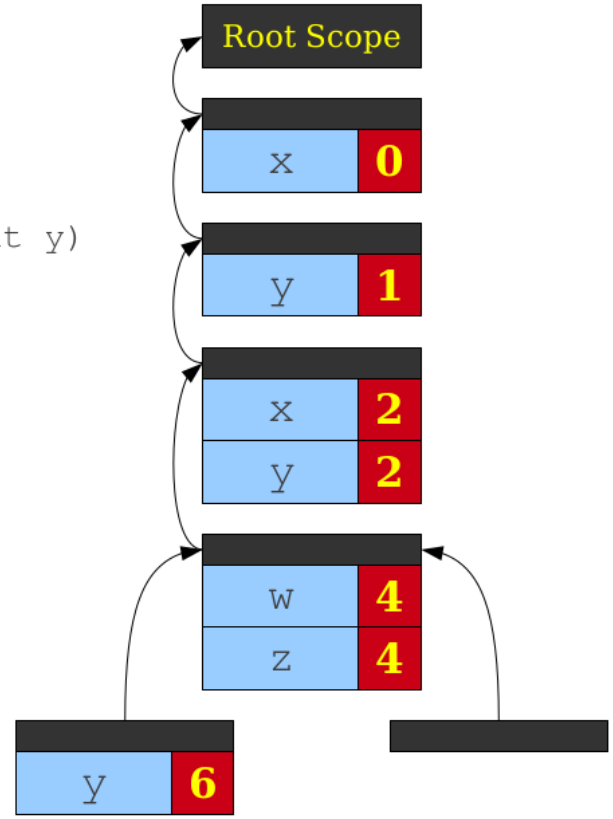
Another View of Symbol Tables

```
 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: }
```



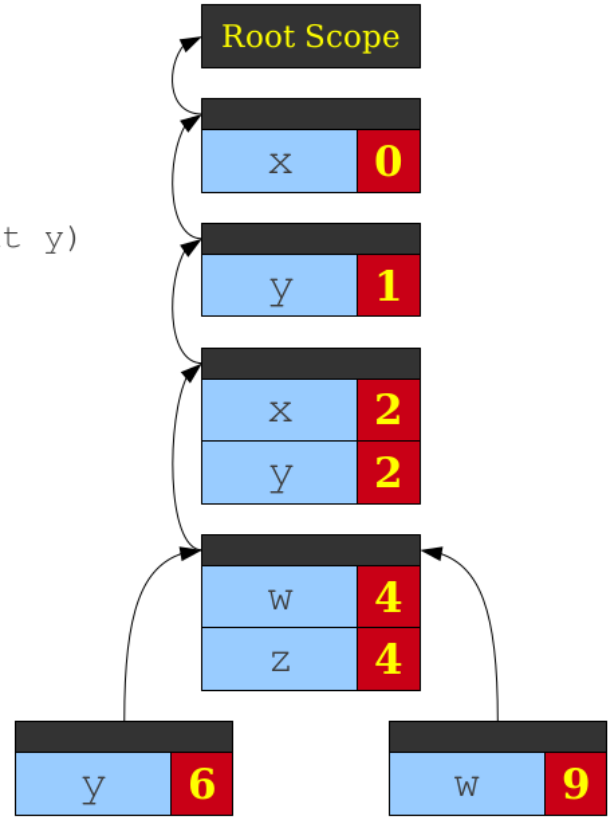
Another View of Symbol Tables

```
 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: }
```



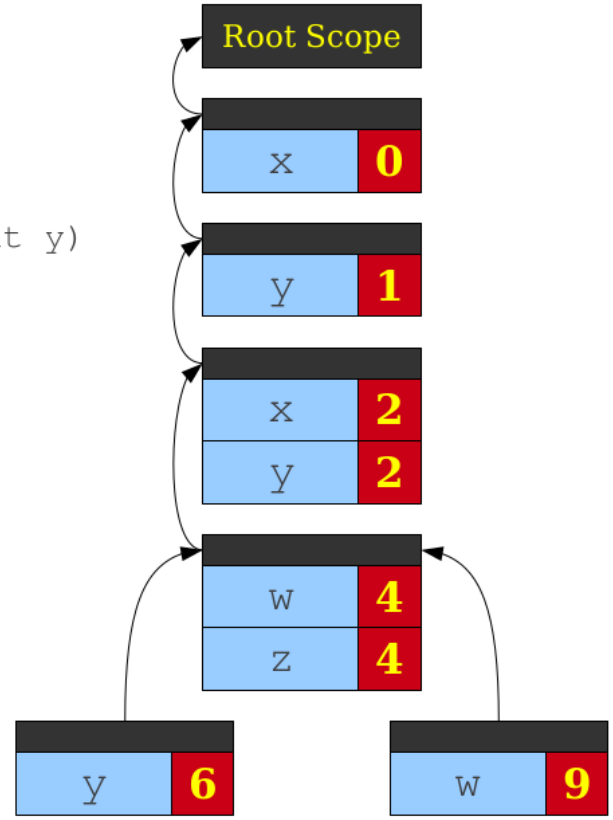
Another View of Symbol Tables

```
 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: }
```



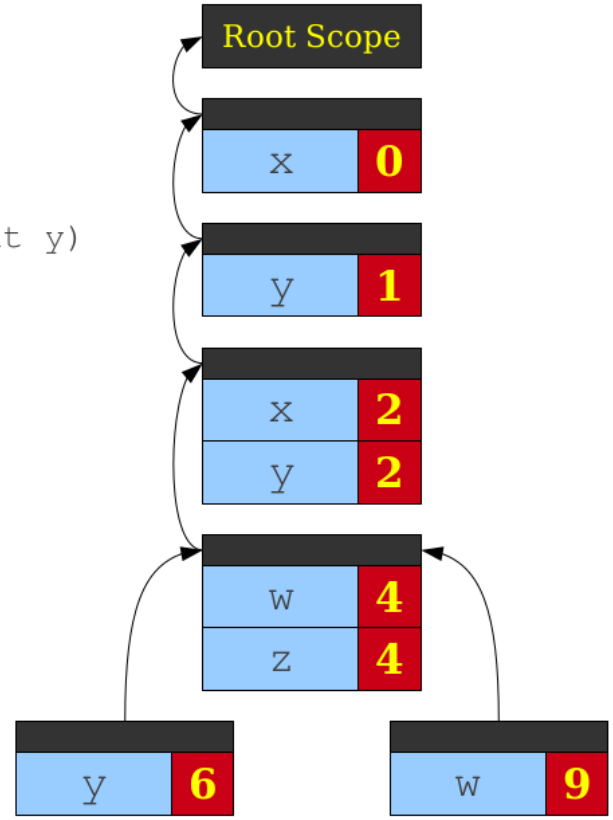
Another View of Symbol Tables

```
 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: }
```



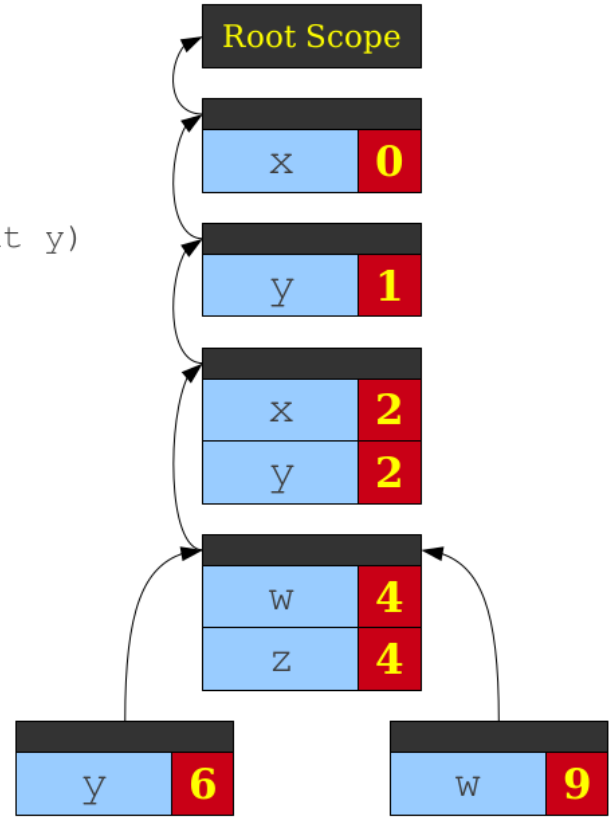
Another View of Symbol Tables

```
 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

```
 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.

Symbol Table Data Structure: Overview

- Unordered List
- Ordered List
- Binary Search Tree
- Hash Table

Symbol Table Data Structure: Overview

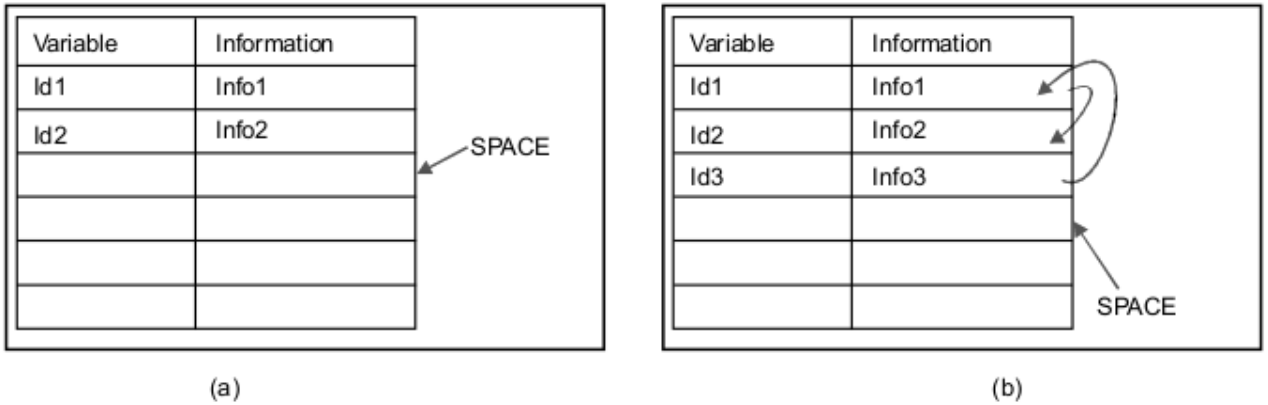
- Unordered List
  - Simplest to implement
  - Implemented as an array or a linked list
  - Linked list can grow dynamically – alleviates problem of a fixed size array
  - ’Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

Variable	Information (type	Space (byte)
a	Integer	2
b	Float	4
c	Character	1
d	Long	4

SPACE

Symbol Table Data Structure: Overview

- Ordered List
  - If an array is sorted, it can be searched using binary search – O(log 2 n)
  - ’Insertion into a sorted array is expensive – O(n) on average
  - It can be considered as a self organizing list

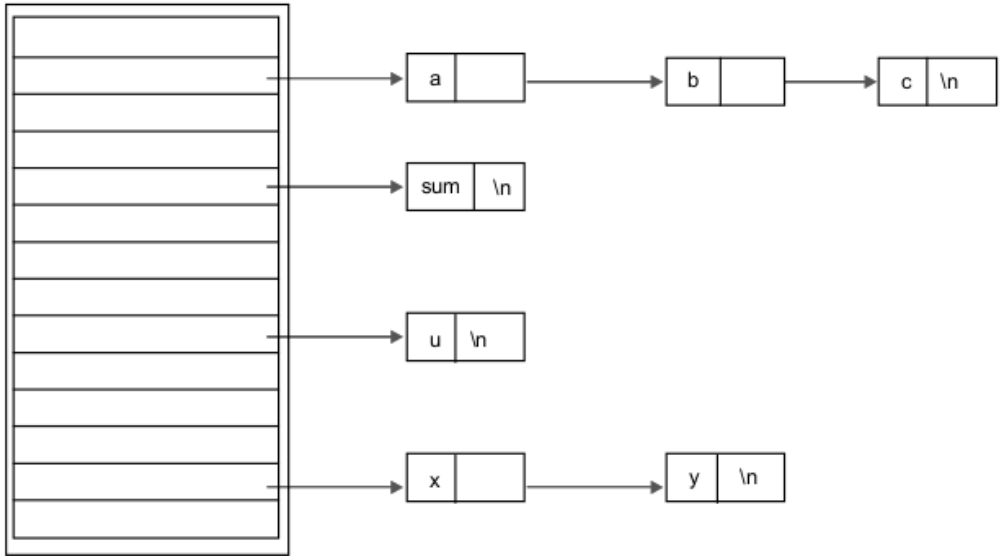


Symbol Table Data Structure: Overview

- Binary Search Tree
  - Can grow dynamically
  - Insertion and lookup are O(log 2 n) on average

Symbol Table Data Structure: Overview

- Hash Table
  - A hash table is an array with index range: 0 to TableSize – 1
  - Most commonly used data structure to implement symbol tables
  - Insertion and lookup can be made very fast – O(1)
  - A hash function maps an identifier name into a table index



Outline

- Introduction
- **Scope-Checking**
  - **Scoping in Object-Oriented Languages**
  - Single- and Multi-Pass Compilers
  - Dynamic and Static Scoping
- Type-Checking

Scoping with Inheritance

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

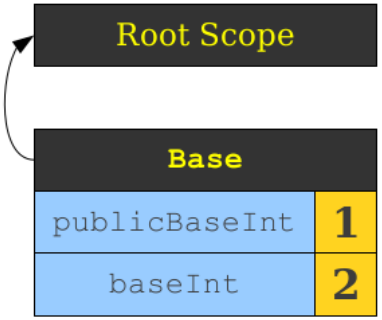
Scoping with Inheritance

Root Scope

```
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;
}
```



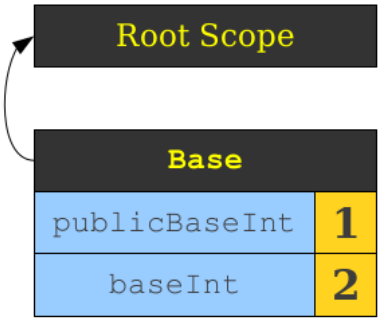
Scoping with Inheritance

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

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);
    }
}
```



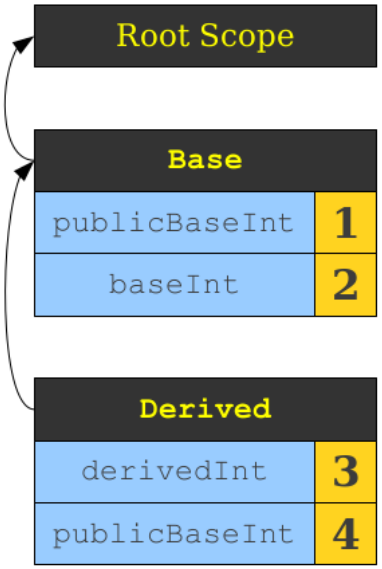
Scoping with Inheritance

```
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

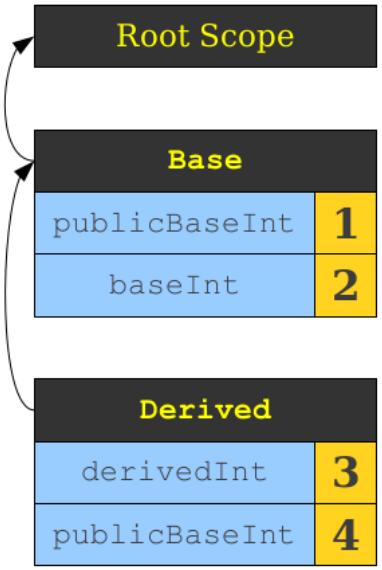
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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

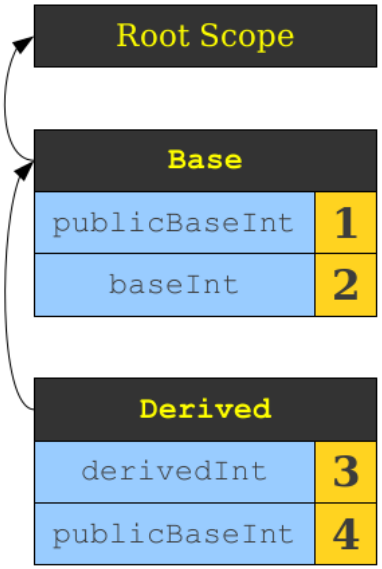
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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);
    }
}
```

```
>
```



Scoping with Inheritance

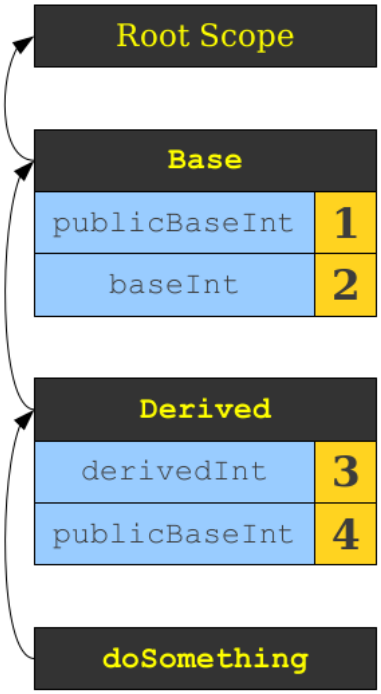
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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

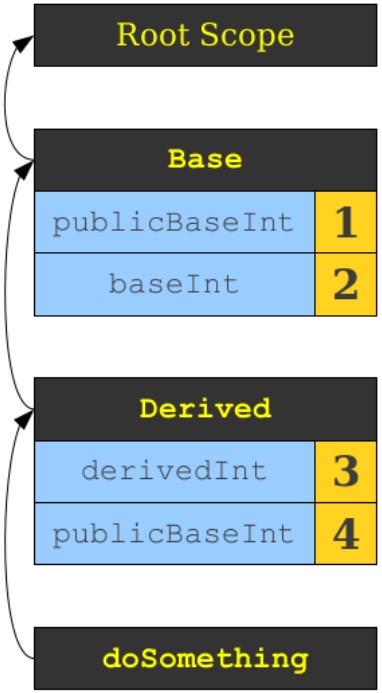
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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

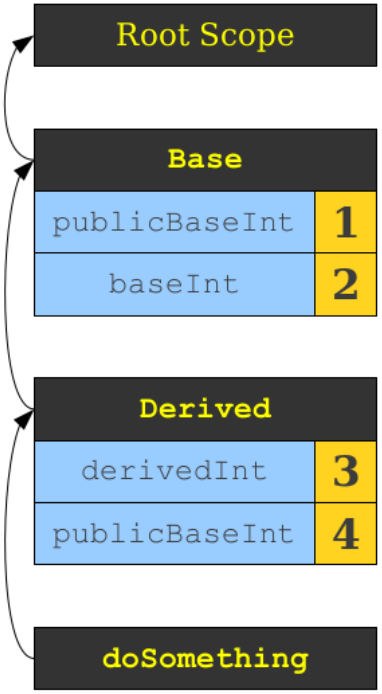
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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

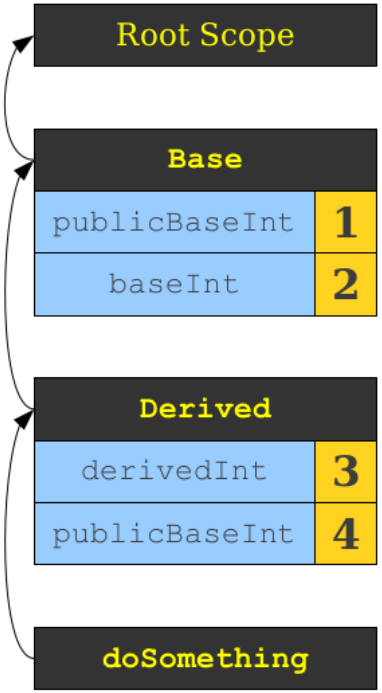
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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
```



Scoping with Inheritance

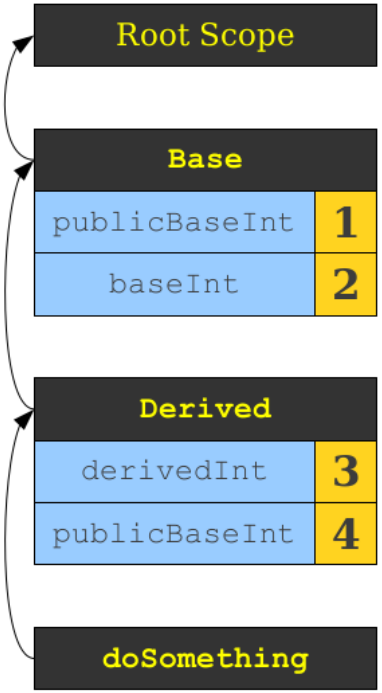
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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

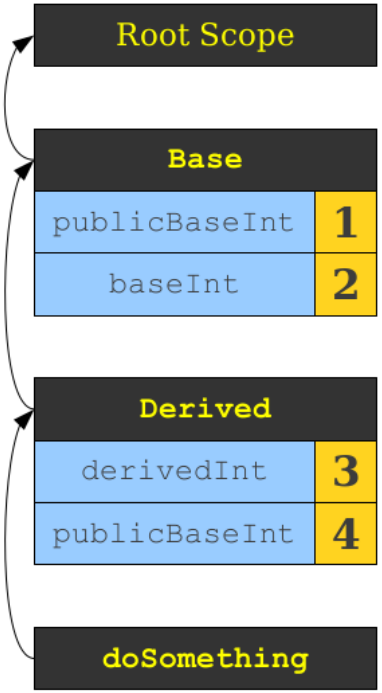
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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
```



Scoping with Inheritance

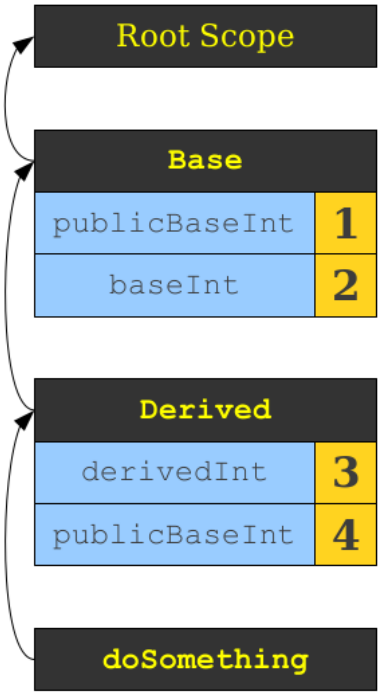
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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

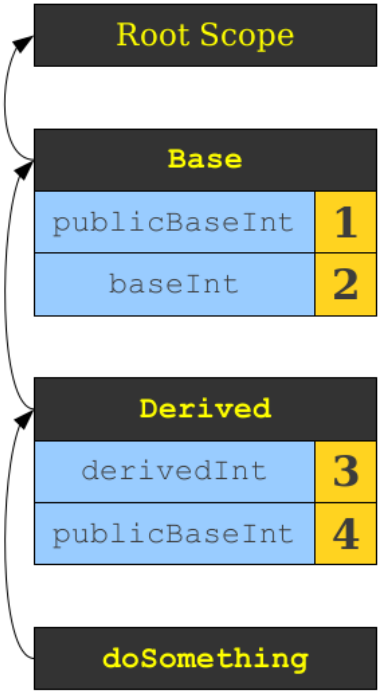
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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

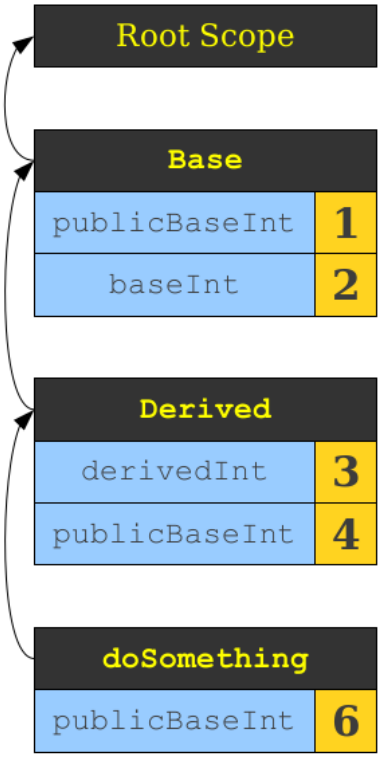
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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
```



Scoping with Inheritance

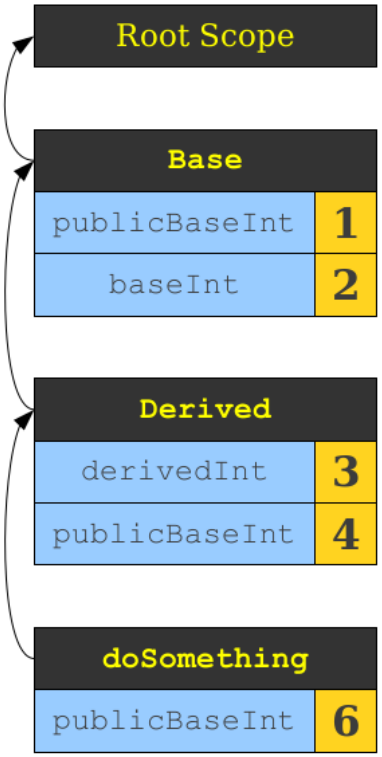
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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

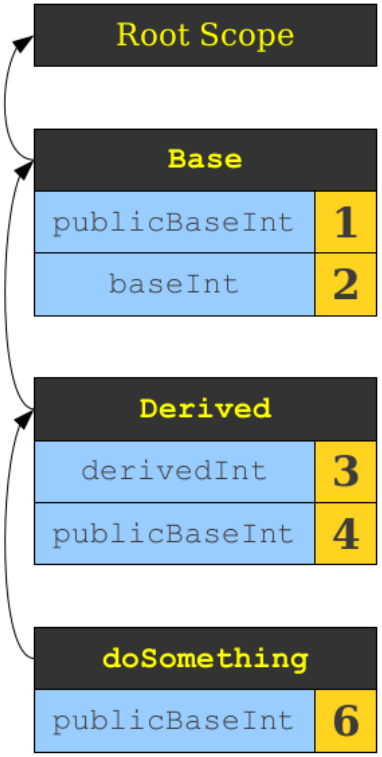
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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
```



Scoping with Inheritance

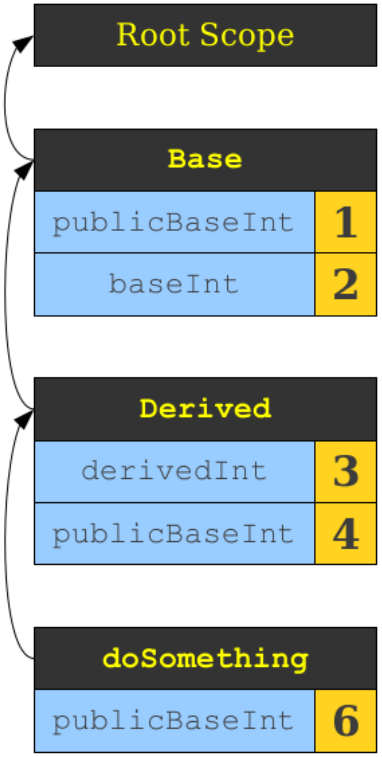
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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
```



Scoping with Inheritance

- Typically, the scope for a derived class will store a link to the scope of its base class.

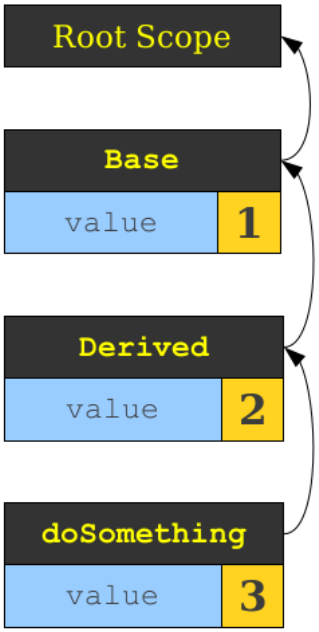
- Looking up a field of a class traverses the scope chain until that field is found or a semantic error is found.

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);
    }
}
```

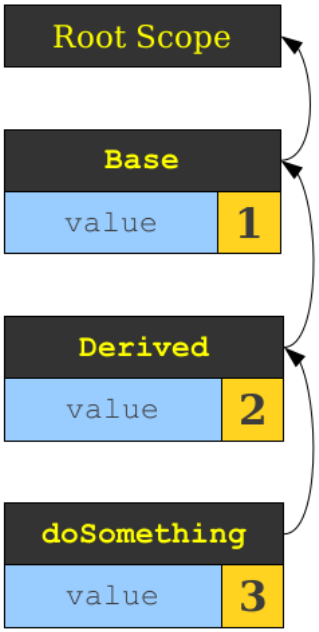


Explicit Disambiguation

```
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);
    }
}
```

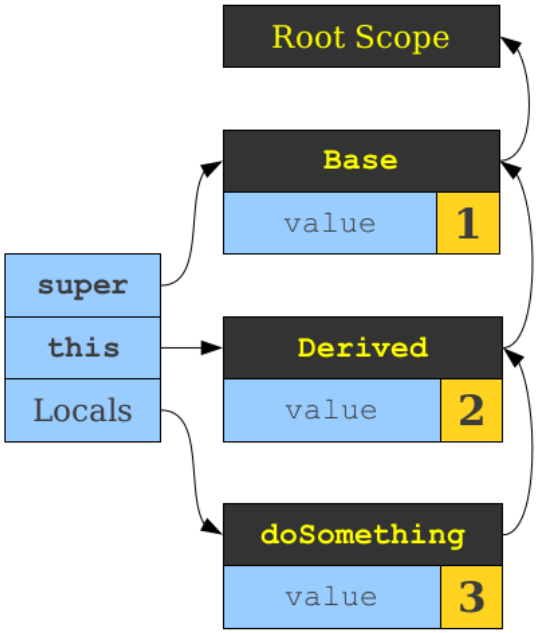


Explicit Disambiguation

```
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);
    }
}
```

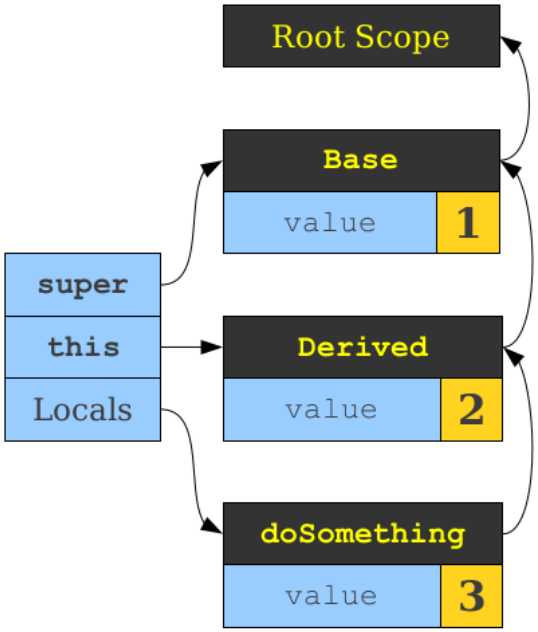


Explicit Disambiguation

```
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);
    }
}
```

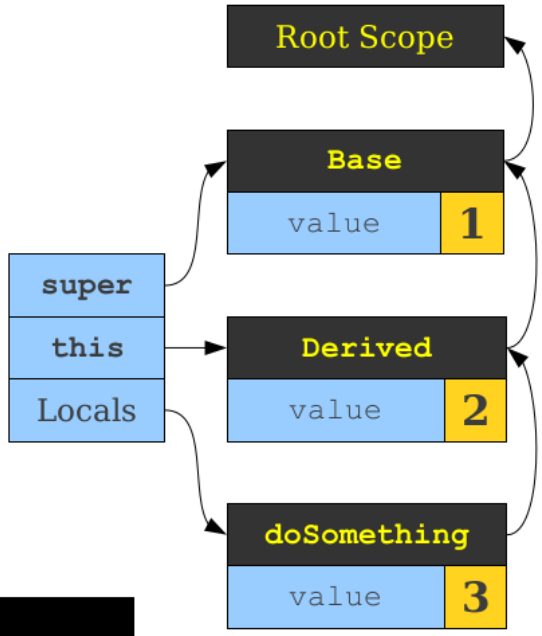


Explicit Disambiguation

```
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);
    }
}
```



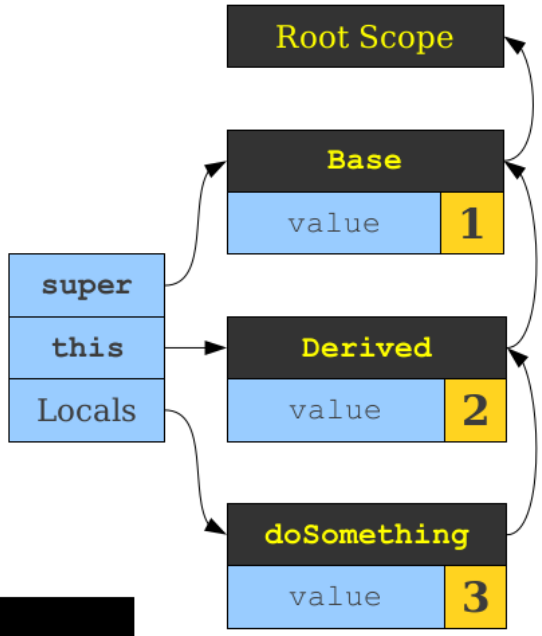

```
>
```

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);
    }
}
```



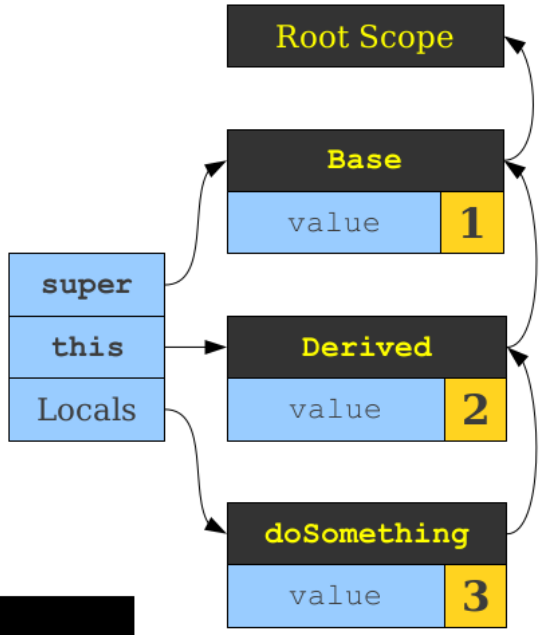

```
>
```

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
```

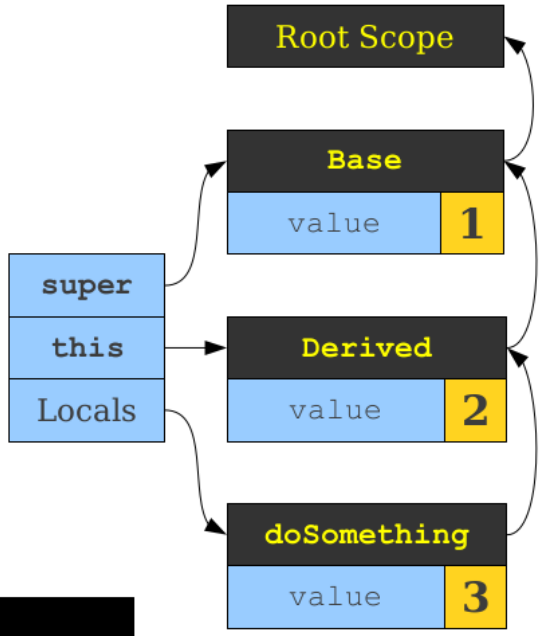
Explicit Disambiguation

```
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
```

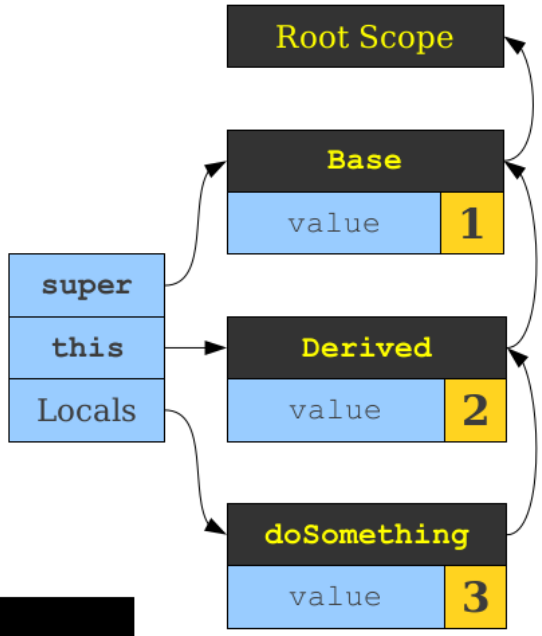


Explicit Disambiguation

```
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);
    }
}
```



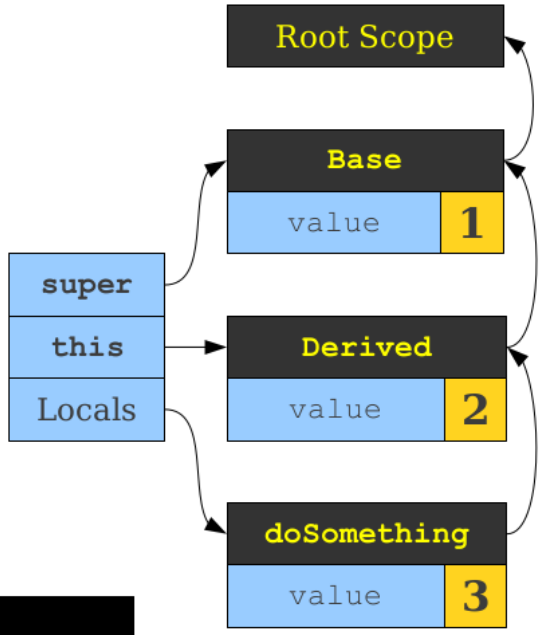

```
> 3
  2
```

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);
    }
}
```



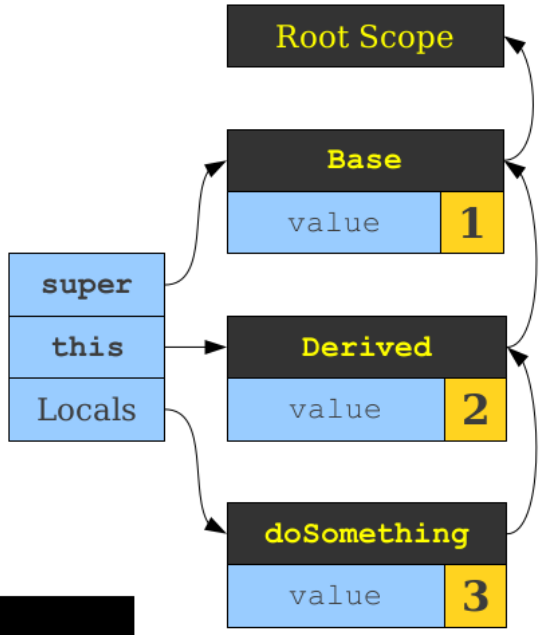

```
> 3
  2
```

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);
    }
}
```



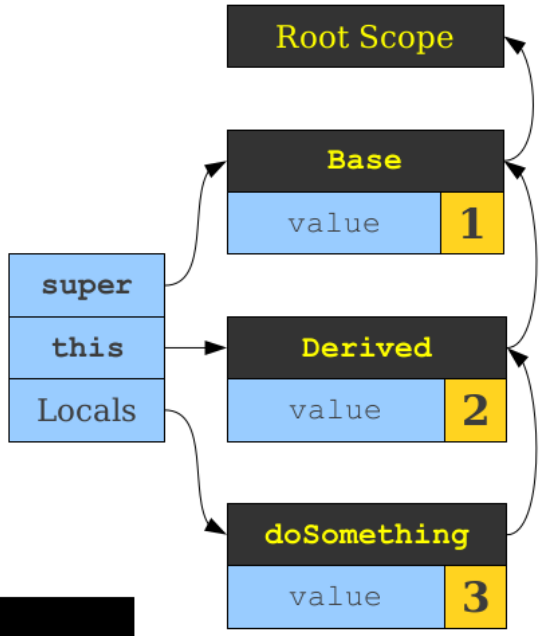

```
> 3
  2
  1
```

Explicit Disambiguation

```
public class Base {
    public int value = 1;
}

public class Derived extends Base {
    public int value = 2;

    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
```



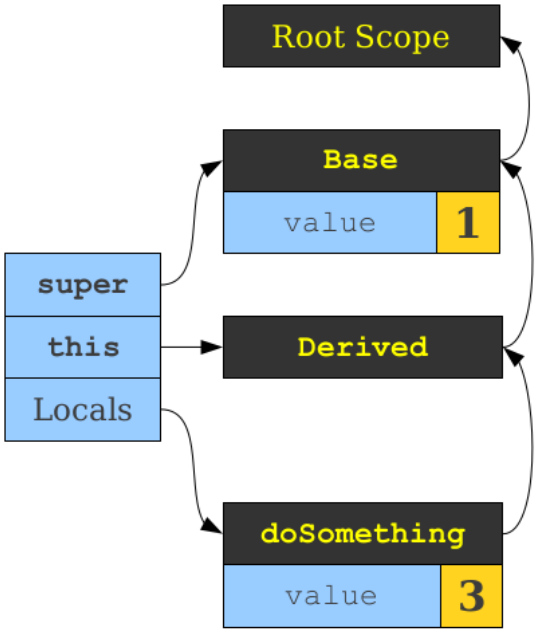

```
> 3
  2
  1
```

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);
    }
}
```

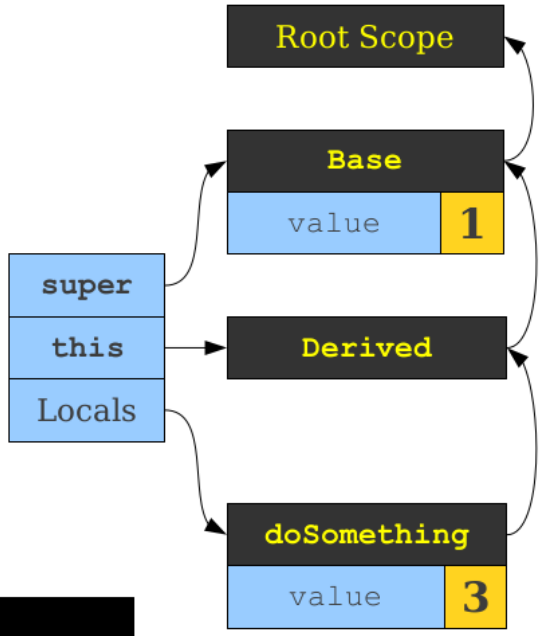


Explicit Disambiguation

```
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);
    }
}
```



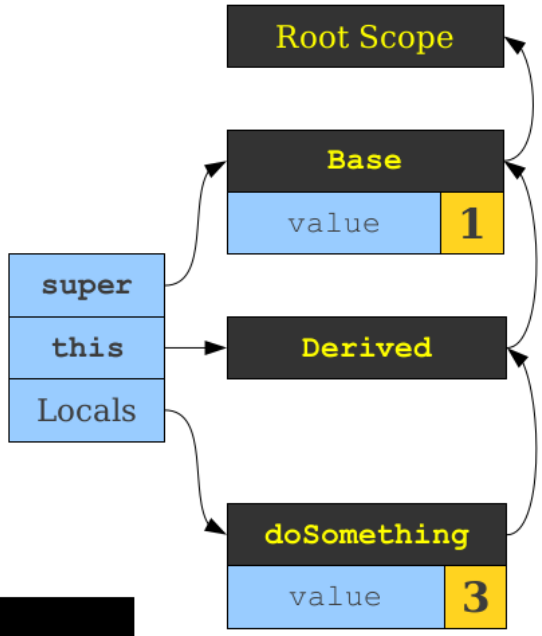

```
>
```

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);
    }
}
```



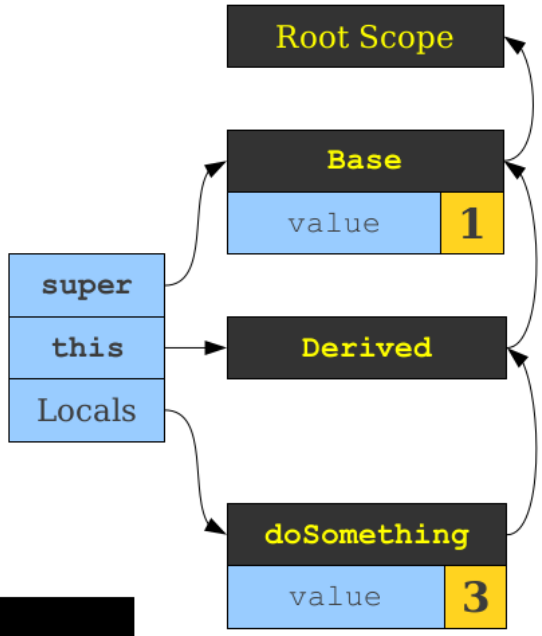
Explicit Disambiguation

```
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
```



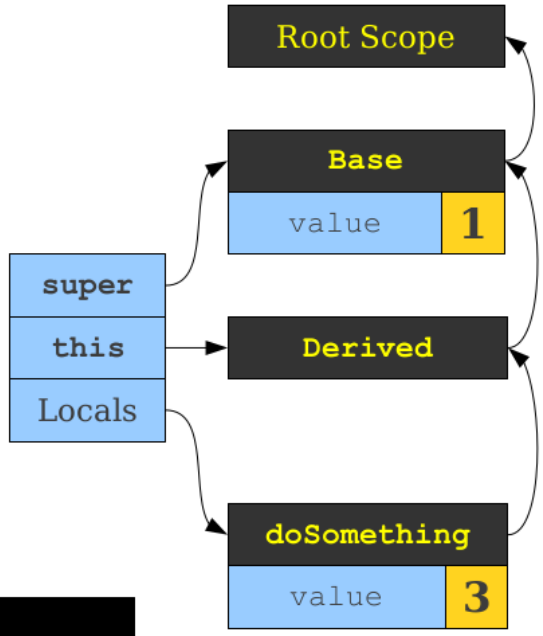
Explicit Disambiguation

```
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
```

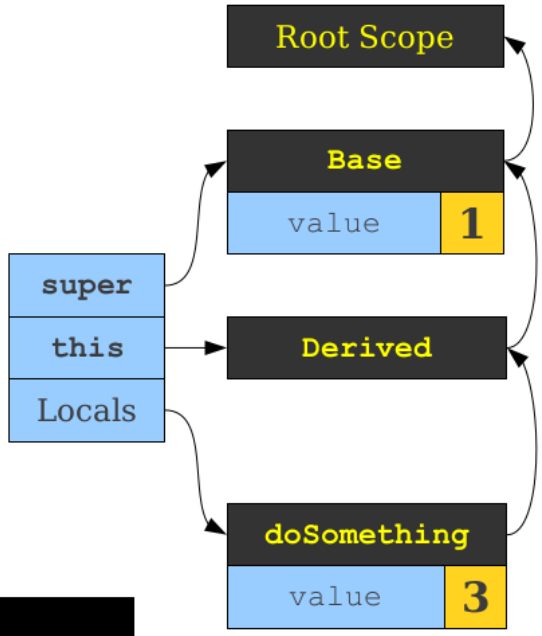


Explicit Disambiguation

```
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);
    }
}
```



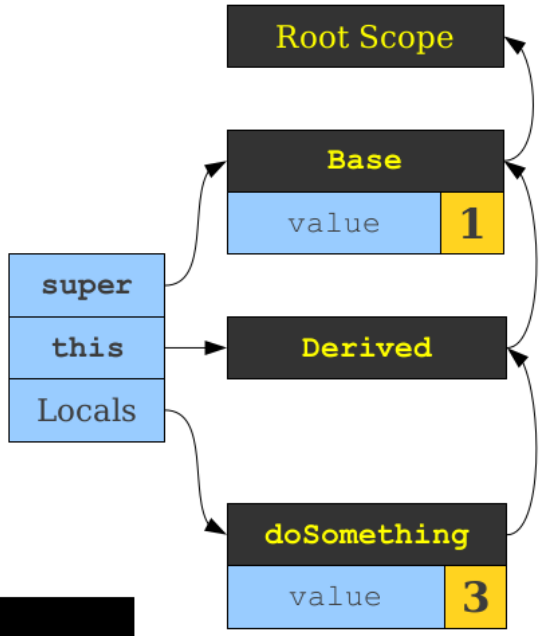

```
> 3
  1
```

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);
    }
}
```



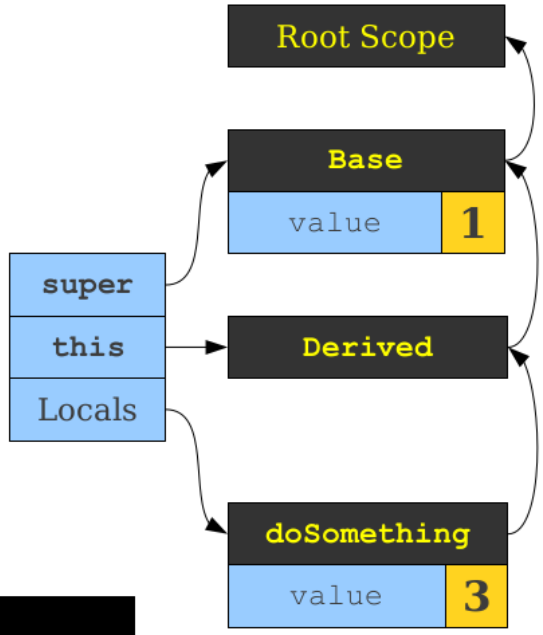

```
> 3
  1
  1
```

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++).

Outline

- Introduction
- **Scope-Checking**
  - Scoping in Object-Oriented Languages
  - **Single- and Multi-Pass Compilers**
  - Dynamic and Static Scoping
- Type-Checking

Scoping in Practice

- Scoping in C++ and Java

```
class A {
public:
/* … */

private:
    B* myB
};

class B {
public:
/* … */

private:
A* myA;
};
```

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

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

Scoping in Practice

- 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), LALR(1), etc.

- Since we only need one token of lookahead, we can do scanning and parsing simultaneously in one pass over the file.

Single- and Multi-Pass Compilers

- 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.

Outline

- Introduction
- **Scope-Checking**
  - Scoping in Object-Oriented Languages
  - Single- and Multi-Pass Compilers
  - **Dynamic and Static Scoping**
- Type-Checking

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 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.

Outline

- Introduction
- Scope-Checking
- **Type-Checking**
  - Static and Dynamic Type Checking
  - Type Checking and Type Inference
  - Types as a Proof System
  - Decaf: Strong Typing

A Short Sample Program

```
class MyClass implements MyInterface {
    string myInteger;

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

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

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

A Short Sample Program

```
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x;
        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

A Short Sample Program

```
class MyClass implements MyInterface {
    string myInteger;

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

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

Wrong type

Can't multiply strings

Variable not declared

Can't redefine functions

Can't add void

No main function

A Short Sample Program

```
class MyClass implements MyInterface {
    string myInteger;

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

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

Wrong type

Can't multiply strings

Variable not declared

Can't add void

No main function

A Short Sample Program

```
class MyClass implements MyInterface {
    string myInteger;

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

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

Wrong type

Can't multiply strings

Can't add void

No main function

A Short Sample Program

```
class MyClass implements MyInterface {
    string myInteger;

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

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

Wrong type

Can't multiply strings

Can't add void

What Remains to Check?

- Type errors.
  - What are types?
  - What is type-checking?
  - A type system for Decaf.

What is a Type?

- This is the subject of some debate.
- To quote Alex Aiken:
  - "The notion varies from language to language.
  - The consensus:
    - A set of values.
    - A set of operations on those values"
- Type errors arise when operations are performed on values that do not support that operation.

Why Do We Need Type Systems?

- Consider the assembly language fragment

```
add $r1, $r2, $r3
```

- What are the types of $r1, $r2, $r3?

Types and Operations

- Certain operations are legal for values of each type
  - It doesn’t make sense to add a function pointer and an integer in C
  - It does make sense to add two integers
  - But both have the same assembly language implementation!

Type Systems

- A language's type system specifies which operations are valid for which types
- The goal of type checking is to ensure that operations are used with the correct types
  - Enforces intended interpretation of values, because nothing else will!

Type Systems

- Strong type systems are systems that never allow for a type error.
  - Java, Python, JavaScript, LISP, Haskell, etc.
- Weak type systems can allow type errors at runtime.
  - C, C++

Outline

- Introduction
- Scope-Checking
- **Type-Checking**
  - **Static and Dynamic Type Checking**
  - Type Checking and Type Inference
  - Types as a Proof System
  - Decaf: Strong Typing

Types of Type-Checking

- Static type checking
  - Analyze the program during compile-time to prove the absence of type errors.
  - Never let bad things happen at runtime.
- Dynamic type checking
  - Check operations at runtime before performing them.
  - More precise than static type checking, but usually less efficient.
- No type checking

The Type Wars

- Competing views on static vs. dynamic typing
- Static typing:
  - Static checking catches many programming errors at compile time
  - Avoids overhead of runtime type checks
- Dynamic typing proponents say:
  - Static type systems are restrictive
  - Rapid prototyping difficult within a static type system
- Dynamic type systems make it easier to prototype; static type systems have fewer bugs.

The Type Wars

- Strongly-typed languages are more robust, weakly-typed systems are often faster.

- Endless debate about what the “right” system is.

The Type Wars

- Strongly-typed languages are more robust, weakly-typed systems are often faster.
- Endless debate about what the “right” system is.
- We are staying out of this!

Our Focus

- Static type checking
  - Type-checking occurs at compile-time.

- Considering class-based inheritance.
- Distinguishing primitive types and classes.

Types

- The user declares types for identifiers
- The compiler infers types for expressions
  - Infers a type for every expression

Outline

- Introduction
- Scope-Checking
- **Type-Checking**
  - Static and Dynamic Type Checking
  - **Type Checking and Type Inference**
  - Types as a Proof System
  - Decaf: Strong Typing

Type Checking and Type Inference

- Type Checking is the process of verifying fully typed programs
- Type Inference is the process of filling in missing type information
- The appropriate formalism for type checking is logical rules of inference

Rules of Inference

- We have seen two examples of formal notation specifying parts of a compiler
  - Regular expressions
  - Context-free grammars

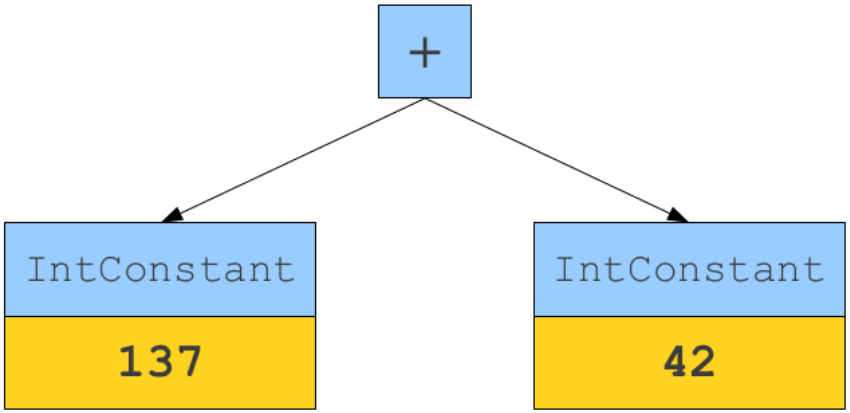
- The appropriate formalism for type checking is logical rules of inference

Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as logical inference.

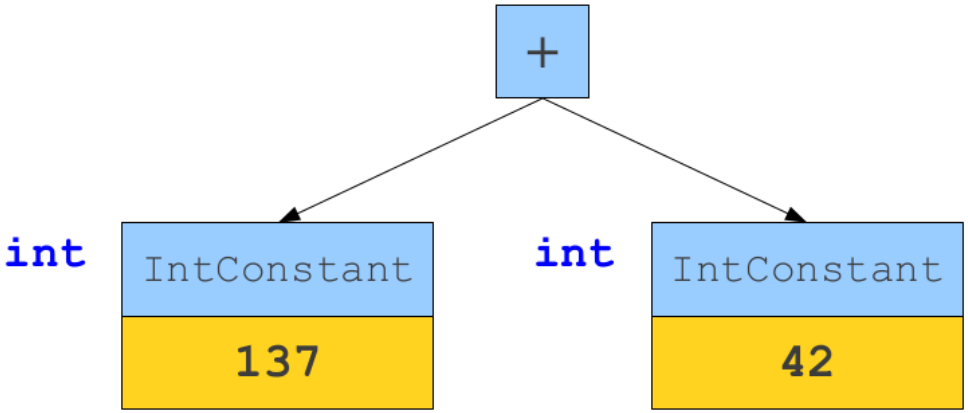
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as logical inference.



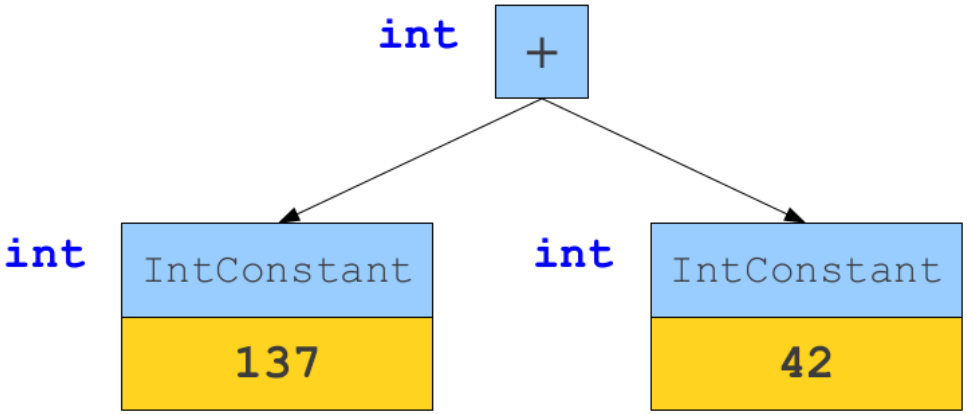
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as logical inference.



Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as logical inference.

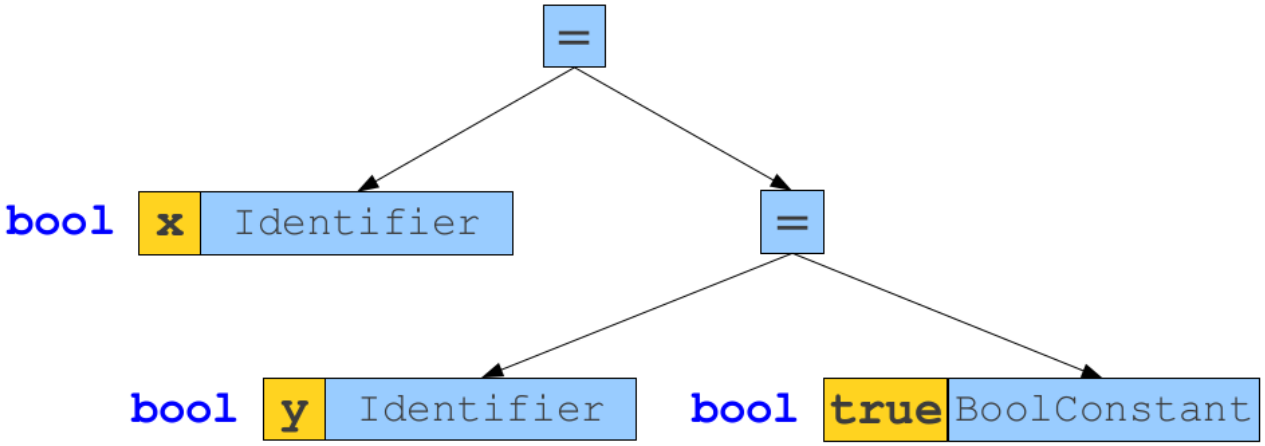


Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as logical inference.

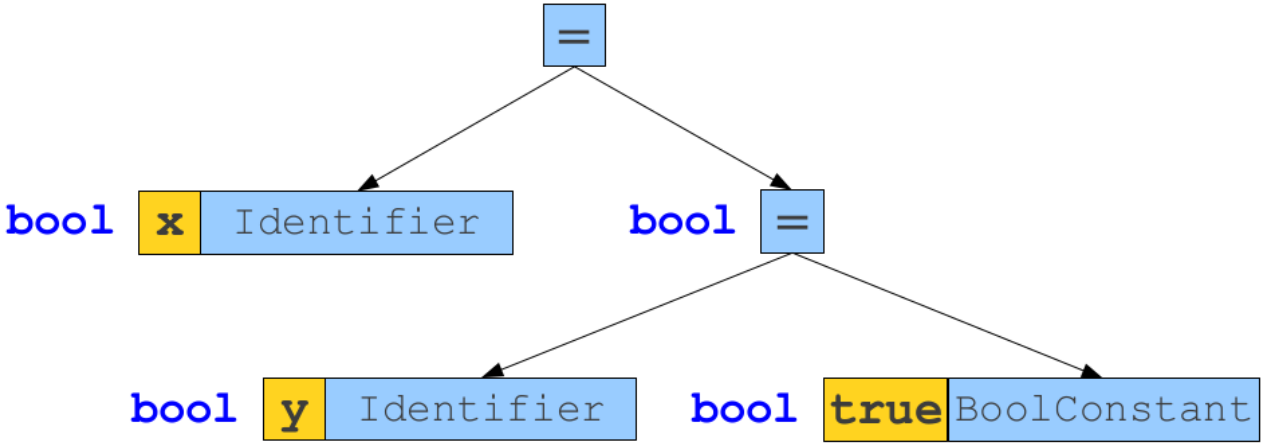
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as logical inference.



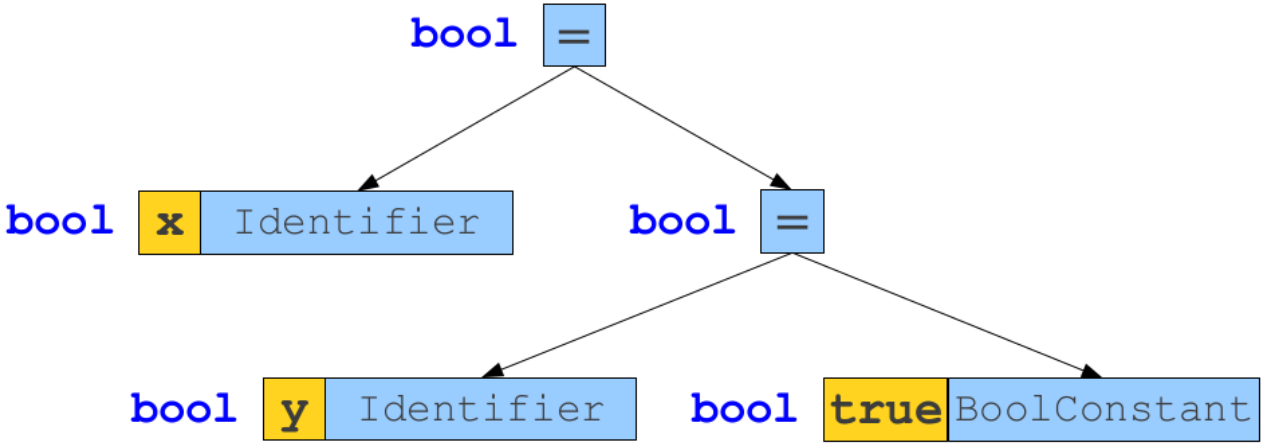
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as logical inference.



Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as logical inference.



Outline

- Introduction
- Scope-Checking
- **Type-Checking**
  - Static and Dynamic Type Checking
  - Type Checking and Type Inference
  - **Types as a Proof System**
  - Decaf: Strong Typing

Type Checking as Proofs

- We can think of syntax analysis as proving claims about the types of expressions.
- We begin with a set of axioms, then apply our inference rules to determine the types of expressions.
- Many type systems can be thought of as proof systems.

Type Checking as Proofs

- Type checking proves facts e:
  - Proof is on the structure of the AST
  - Proof has the shape of the AST
  - One type rule is used for each AST node
- In the type rule used for a node e:
  - Hypotheses are the proofs of types of e' s subexpressions
  - Conclusion is the type of e
- Types are computed in a bottom-up pass over the AST

Why Rules of Inference?

- Inference rules have the form

  If Hypothesis is true, then Conclusion is true

- Type checking computes via reasoning

  If E 1 and E 2 have certain types, then E 3 has a certain type

- Rules of inference are a compact notation for *"If-Then"* statements

From English to an Inference Rule

- The notation is easy to read with practice
- Start with a simplified system and gradually add features
- Building blocks
  - Symbol $\wedge$ is "and"
  - Symbol $\Rightarrow$ is "if-then"
  - x:T is "x has type T"

From English to an Inference Rule

If e 1 has type Int and e 2 has type Int,

then e 1 + e 2 has type Int

(e 1 has type Int ∧ e 2 has type Int) ⇒

e 1 + e 2 has type Int

(e 1 : Int ∧ e 2 : Int) ⇒ e 1 + e 2 : Int

From English to an Inference Rule

- The statement (e 1 : Int ∧ e 2 : Int) ⇒ e 1 + e 2 : Int is a special case of

  Hypothesis 1 ∧ . . . ∧ Hypothesis n ⇒ Conclusion

- This is an inference rule

- By tradition inference rules are written

```
⊢ Hypothesis ... ⊢ Hypothesis
-------------------------------------
⊢ Conclusion
```

Example of Inference Rules

$$\frac{\vdash e_1 : \texttt{int} \qquad \vdash e_2 : \texttt{int}}{\vdash e_1 + e_2 : \texttt{int}}$$

$$\frac{\vdash e_1 : \texttt{double} \qquad \vdash e_2 : \texttt{double}}{\vdash e_1 + e_2 : \texttt{double}}$$

Example of Inference Rules

$$\frac{\vdash e_1 : T \quad \vdash e_2 : T \quad T \text{ is a primitive type}}{\vdash e_1 \texttt{ == } e_2 : \texttt{bool}}$$

$$\frac{\vdash e_1 : T \quad \vdash e_2 : T \quad T \text{ is a primitive type}}{\vdash e_1 \texttt{ != } e_2 : \texttt{bool}}$$

Outline

- Introduction
- Scope-Checking
- **Type-Checking**
  - Static and Dynamic Type Checking
  - Type Checking and Type Inference
  - Types as a Proof System
  - **Decaf: Strong Typing**

The Decaf Language

- Decaf is a strongly-typed, object-oriented language with support for inheritance and encapsulation.
- By design, it has many similarities with C/C++/Java
  - It is not an exact match to any of those languages.
- The feature set has been trimmed down and simplified to keep the programming projects manageable.
- The language is still expressive enough to build all sorts of nifty object-oriented programs.

Static Typing in Decaf

- Static type checking in Decaf consists of two separate processes:
  - Inferring the type of each expression from the types of its components.
  - Confirming that the types of expressions in certain contexts matches what is expected.
- Logically two steps, but you will probably combine into one pass.

Summary

- The importance of semantic analysis
- Main tasks in semantic analysis
  - Scope-checking
  - Type-checking

Question?