The University for
world-class professionals

Manchester
Metropolitan
University

Compilers: Semantic Analysis

Dr Paris Yiapanis
room: John Dalton E151
email: p.yiapanis@mmu.ac.uk

Where we are…

- ~~Admin and overview~~
- ~~Lexical analysis~~
- ~~Parsing~~
- **Semantic analysis**
- Machine-independent optimisation
- Code generation
- Hardware architectures
- Machine-dependent optimisation
- Review

Objectives

- Introduce *semantic analysis*
  (also known as *context-sensitive analysis*)
- Motivate the need for *intermediate representations*
- Introduce the concept of a *symbol table*
- Consider *scope* in detail

Quiz!

```
1    fie(a, b, c, d) {
2       int a, b, c, d;
3       …
4    }
5
6    fee() {
7       int f[3], g[0], h, i, j, k;
8       char *p;
9
10      fie(h, i, “ab”, j, k);
11      k = f * i + j;
12      h = g[17];
13      printf(“<%s,%s>.\n”, p, q);
14      p = 10;
15   }
```

Find the semantic errors. (At least 6; there are 7 errors and one possible error.)

This is C code. You should have enough experience with different languages to identify problems here even if you don’t know C – with the possible exception of the statement at line 13.

Quiz!

```
fie(a, b, c, d) {
    int a, b, c, d;
    …
}

fee() {
    int f[3], g[0], h, i, j, k;
    char *p;

    fie(h, i, “ab”, j, k);
    k = f * i + j;
    h = g[17];
    printf(“<%s,%s>.\n”, p, q);
    p = 10;
}
```

types of parameters not specified

local declarations make function arguments inaccessible

zero-length array?

5 arguments passed to function with only 4 parameters

incompatible types (array of int * int)

access beyond array bounds

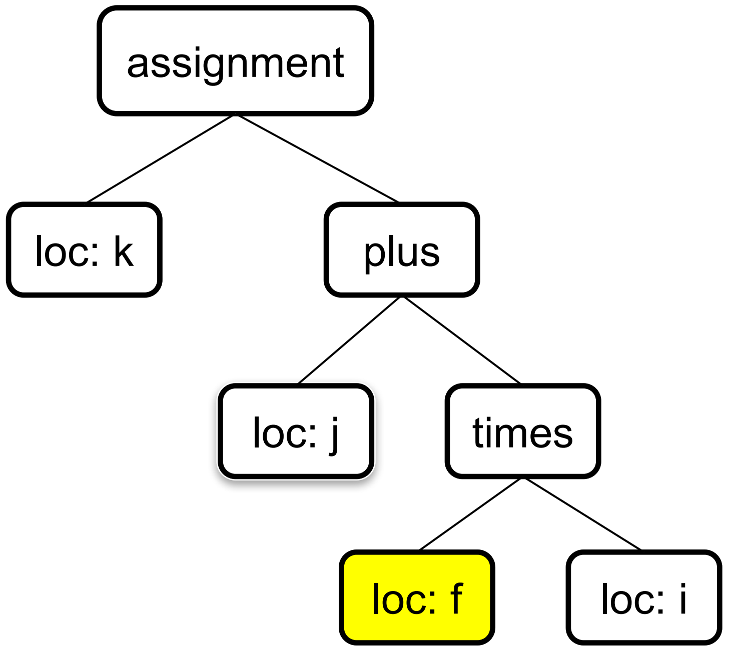
q has not been declared

incompatible assignment type?

Context-Sensitive Analysis

- Also referred to as *semantic analysis* or *semantic elaboration*
- Recall the first two phases of the compiler:
  - Lexical analysis or scanning: identify the tokens in the input, pass them to the
  - Parser or syntactical analysis: ensure tokens are arranged in a grammatically-correct manner, build a *syntax-related tree*
- Grammatically correct isn't good enough to generate working machine code – tree needs further analysis

Syntax Tree for line 11: k = f * i + j;



- *Syntactically*, this statement is fine
- It is impossible to detect the error without the *context* provided by line 7:

```
int f[3], g[0], h, i, j, k;
```

What Sorts of Semantic Checking?

- Type checking
- Scope checking
- Variables declared before used
- Sanity checking on array bounds
- Must pass right number and types of arguments to methods
- Methods must return a value of the correct type

… this is *not* an exhaustive list!

Could you do these checks while parsing?

- Yes, but context-free grammars by definition are *context-free*
  - Would need to augment grammars with contextual information
    - this is a valid approach. Best known form is use of *attribute grammars* (non-examinable, but see text if interested)
- More common to perform multiple passes of the input, traversing the syntax tree generated by the parser, to perform semantic checking

Semantic Checking in Practice

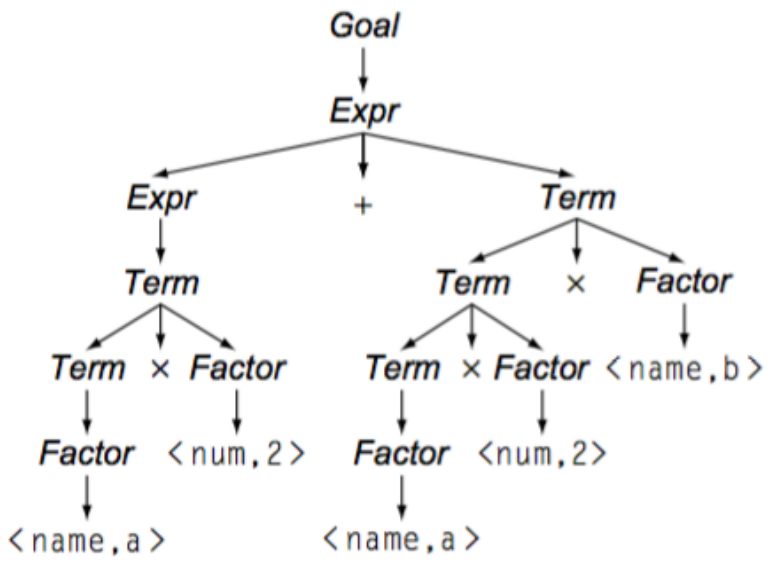
- Implementation of scanning and parsing is heavily theory-driven
- There *are* formal theory-driven approaches to semantic checking
- In practice, ad hoc approaches tend to be used, annotating the syntax tree from the parser and using a *symbol table*

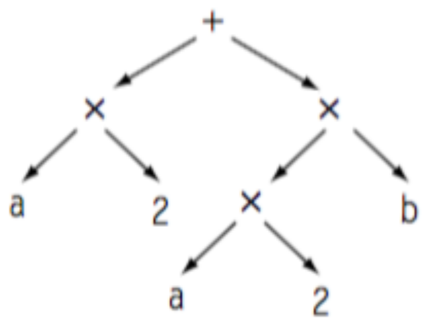
Parse tree Vs Abstract Syntax Tree

- Input: a × 2 + a × 2 × b

Parse tree Vs Abstract Syntax Tree

- Input: a × 2 + a × 2 × b





Parse Tree

Abstract Syntax Tree

Parse tree Vs Abstract Syntax Tree

- Parse trees are large relative to source code
- Parse trees represent the complete derivation
- In theory parse trees make explanations clearer but,
- In most applications more concise versions are used (e.g. AST)

Scope Checking

```
class Main {
   int a=1;
   public static void main(String[] args) {
      int a=2, b=2, c=2;
      Main m = new Main();

      System.out.println(a + " " + b + " " + c);
      m.method(a, b, c);
   }
   public void method(int a, int b, int c) {
      System.out.println(a + " " + b + " " + c);
      System.out.println(this.a + " " + b + " " + c);
   }
}
```

Scope Checking

```
class Main {
    int a=1;
    public static void main(String[] args) {
        int a=2, b=2, c=2;
        Main m = new Main();

        System.out.println(a + " " + b + " " + c);
        m.method(a, b, c);
    }
    public void method(int a, int b, int c) {
        System.out.println(a + " " + b + " " + c);
        System.out.println(this.a + " " + b + " " + c);
    }
}
```

Scope Checking

```
class Main {
    int a=1;
    public static void main(String[] args) {
        int a=2, b=2, c=2;
        Main m = new Main();

        System.out.println(a + " " + b + " " + c);
        m.method(a, b, c);
    }
    public void method(int a, int b, int c) {
        System.out.println(a + " " + b + " " + c);
        System.out.println(this.a + " " + b + " " + c);
    }
}
```

Scope Checking

```
class Main {
    int a=1;
    public static void main(String[] args) {
        int a=2, b=2, c=2;
        Main m = new Main();

        System.out.println(a + " " + b + " " + c);
        m.method(a, b, c);
    }
    public void method(int a, int b, int c) {
        System.out.println(a + " " + b + " " + c);
        System.out.println(this.a + " " + b + " " + c);
    }
}
```

Scope Checking

- It is possible have several identifiers with the same name (e.g. ‘a’ below) but with different scopes:

```
class Main {
   int a=1;
   public static void main(String[] args) {
      int a=2, b=2, c=2;
      Main m = new Main();

      System.out.println(a + " " + b + " " + c);
      m.method(a, b, c);
   }
   public void method(int a, int b, int c) {
      System.out.println(a + " " + b + " " + c);
      System.out.println(this.a + " " + b + " " + c);
   }
}
```

Scope Checking

- It is possible have several identifiers with the same name (e.g. ‘a’ below) but with different scopes:

```
class Main {
    int a=1;
    public static void main(String[] args) {
        int a=2, b=2, c=2;
        Main m = new Main();

        System.out.println(a + " " + b + " " + c);
        m.method(a, b, c);
    }
    public void method(int a, int b, int c) {
        System.out.println(a + " " + b + " " + c);
        System.out.println(this.a + " " + b + " " + c);
    }
}
```

Scope Checking

- It is possible have several identifiers with the same name (e.g. ‘a’ below) but with different scopes:

```
class Main {
    int a=1;
    public static void main(String[] args) {
        int a=2, b=2, c=2;
        Main m = new Main();

        System.out.println(a + " " + b + " " + c);
        m.method(a, b, c);
    }
    public void method(int a, int b, int c) {
        System.out.println(a + " " + b + " " + c);
        System.out.println(this.a + " " + b + " " + c);
    }
}
```

Scope Checking

- It is possible have several identifiers with the same name (e.g. ‘a’ below) but with different scopes:

```
class Main {
   int a=1;
   public static void main(String[] args) {
      int a=2, b=2, c=2;
      Main m = new Main();

      System.out.println(a + " " + b + " " + c);
      m.method(a, b, c);
   }
   public void method(int a, int b, int c) {
      System.out.println(a + " " + b + " " + c);
      System.out.println(this.a + " " + b + " " + c);
   }
}
```

A Symbol Table?

- Say we had an expression such as

  a = a × a + b – c ÷ a
- The AST would be something like this:

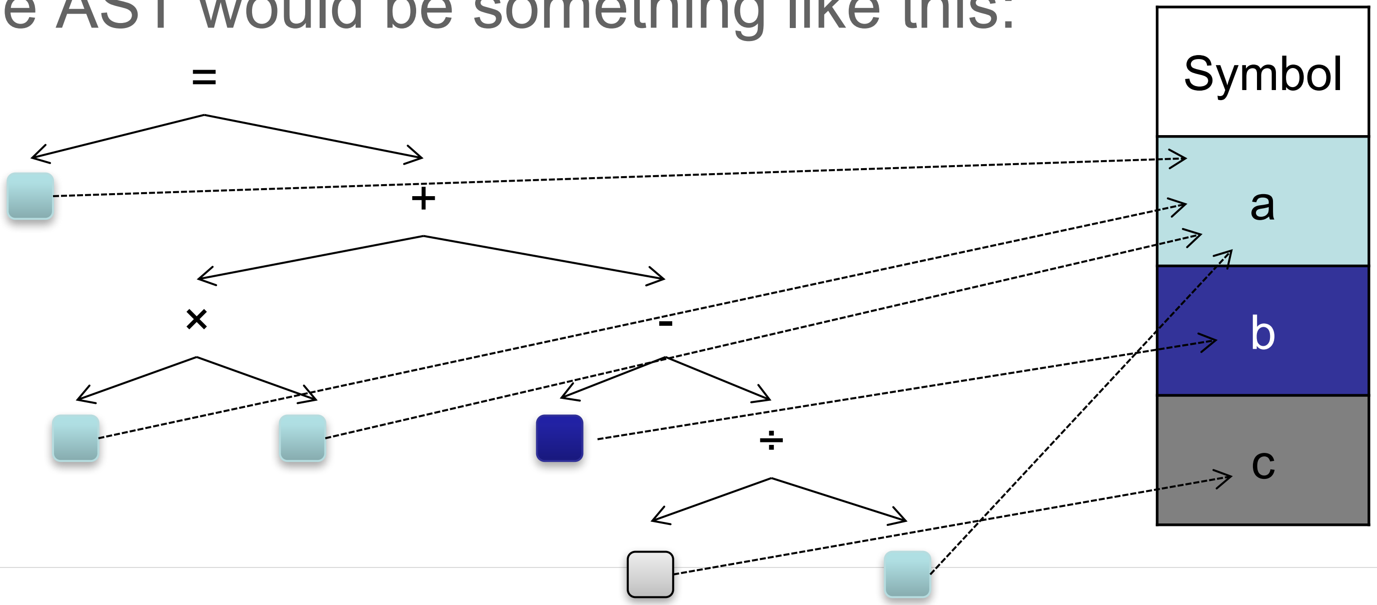
$$\begin{tikzcd}
 & & = \arrow[dll] \arrow[drr] & & & & & \\
\langle\text{name, a}\rangle & & & & + \arrow[dll] \arrow[dr] & & & \\
 & & \times \arrow[dl] \arrow[dr] & & & - \arrow[dl] \arrow[dr] & & \\
 & \langle\text{name, a}\rangle & & \langle\text{name, a}\rangle & \langle\text{name, b}\rangle & & \div \arrow[dl] \arrow[dr] & \\
 & & & & & \langle\text{name, c}\rangle & & \langle\text{name, a}\rangle
\end{tikzcd}$$

Notice how <name, a> occurs multiple times, but is it all a reference to the same thing?

A Symbol Table?

This is a simplification: the symbol table contains more information about each symbol than just the name (e.g. *type*).

- Say we had an expression such as
  a = a × a + b – c ÷ a
- The AST would be something like this:



Scope Checking

```c
int x = 137;
int z = 42;
int MyFunction(int x, int y) {
  printf("%d,%d,%d\n", x, y, z);
  {
    int x, z;
    z = y;
    x = z;
    {
      int y = x;
      {
        printf("%d,%d,%d\n", x, y, z);
      }
      printf("%d,%d,%d\n", x, y, z);
    }
    printf("%d,%d,%d\n", x, y, z);
  }
}
```

This is C code. It allows you to reuse identifier names, with local scope, in blocks. In Java, you can reuse them in classes or methods, but not sub-blocks like this.

Scope Checking

```
int x = 137;
int z = 42;
int MyFunction(int x, int y) {
  printf("%d,%d,%d\n", x, y, z);
  {
    int x, z;
    z = y;
    x = z;
    {
      int y = x;
      {
        printf("%d,%d,%d\n", x, y, z);
      }
      printf("%d,%d,%d\n", x, y, z);
    }
    printf("%d,%d,%d\n", x, y, z);
  }
}
```

Scope Checking

Symbol Table	
Identifier	Declared at

```
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:}
```

Scope Checking

Symbol Table	
Identifier	**Declared at**
Global Scope	

```
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:}
```

Scope Checking

Symbol Table	
Identifier	**Declared at**
Global Scope	
x	0
z	1

```
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:}
```

Manchester Metropolitan University

Scope Checking

Symbol Table	
Identifier	**Declared at**
Global Scope	
x	0
z	1
***Scope 1**: MyFunction parameters Block starting line 2*	
x	2
y	2

```
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:}
```

Scope Checking

Symbol Table	
Identifier	**Declared at**
Global Scope	
x	0
z	1
Scope 1*: MyFunction parameters Block starting line 2*	
x	2
y	2
Scope 2*: Block starting line 4*	
x	5
z	5

```
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:}
```

Scope Checking

Symbol Table	
Identifier	**Declared at**
Global Scope	
x	0
z	1
***Scope 1**: MyFunction parameters Block starting line 2*	
x	2
y	2
***Scope 2**: Block starting line 4*	
x	5
z	5
***Scope 3**: Block starting line 8*	
y	9

```
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:}
```

Scope Checking

Symbol Table	
Identifier	**Declared at**
Global Scope	
x	0
z	1
***Scope 1**: MyFunction parameters Block starting line 2*	
x	2
y	2
***Scope 2**: Block starting line 4*	
x	5
z	5
***Scope 3**: Block starting line 8*	
y	9
***Scope 4**: Block starting line 10*	

```
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:}
```

Scope Checking

Symbol Table	
Identifier	**Declared at**
Global Scope	
x	0
z	1
***Scope 1**: MyFunction parameters Block starting line 2*	
x	2
y	2
***Scope 2**: Block starting line 4*	
x	5
z	5
***Scope 3**: Block starting line 8*	
y	9
***Scope 4**: Block starting line 10*	

```
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:}
```

Scope 0

Scope 1

Scope 2

Scope 3

Scope 4

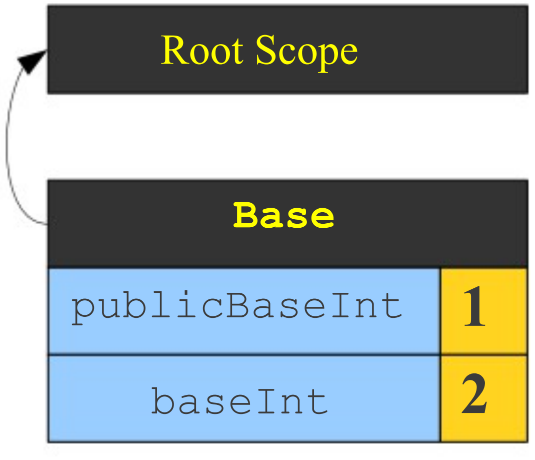
Scoping with Inheritance

Root Scope

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

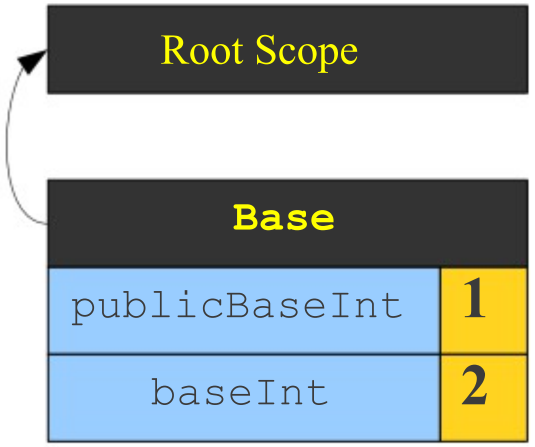
Scoping with Inheritance

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



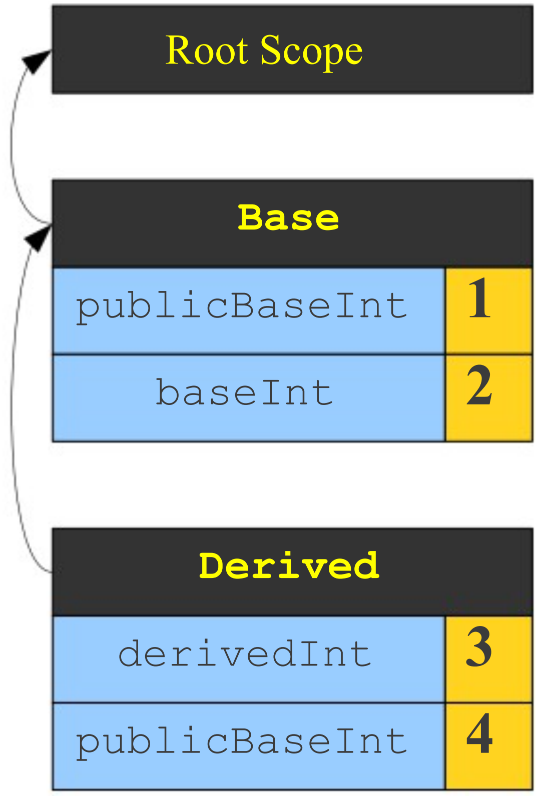
Scoping with Inheritance

```
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

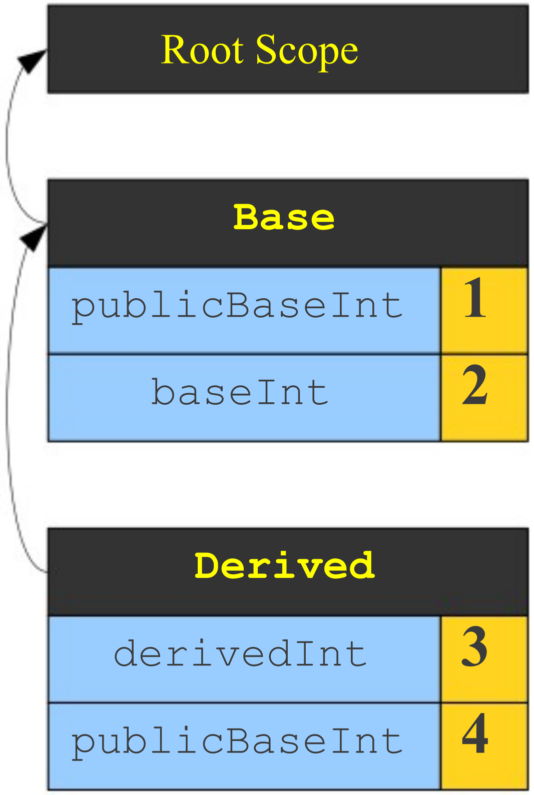
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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);
    }
}
```

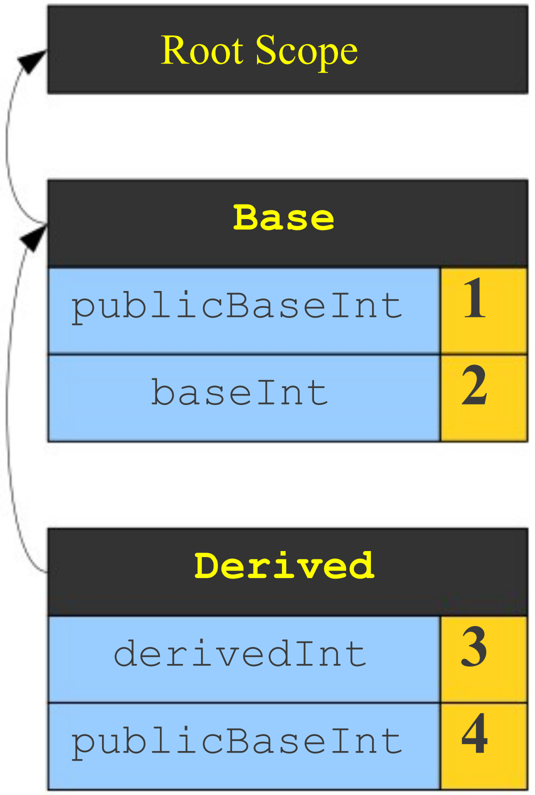
```
>
```



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);
    }
}
```

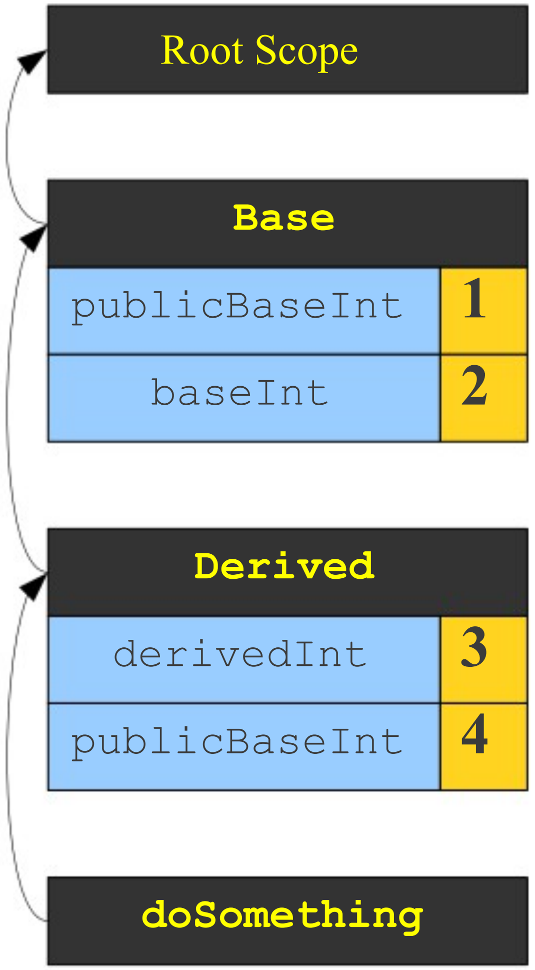
```
>
```



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);
    }
}
```

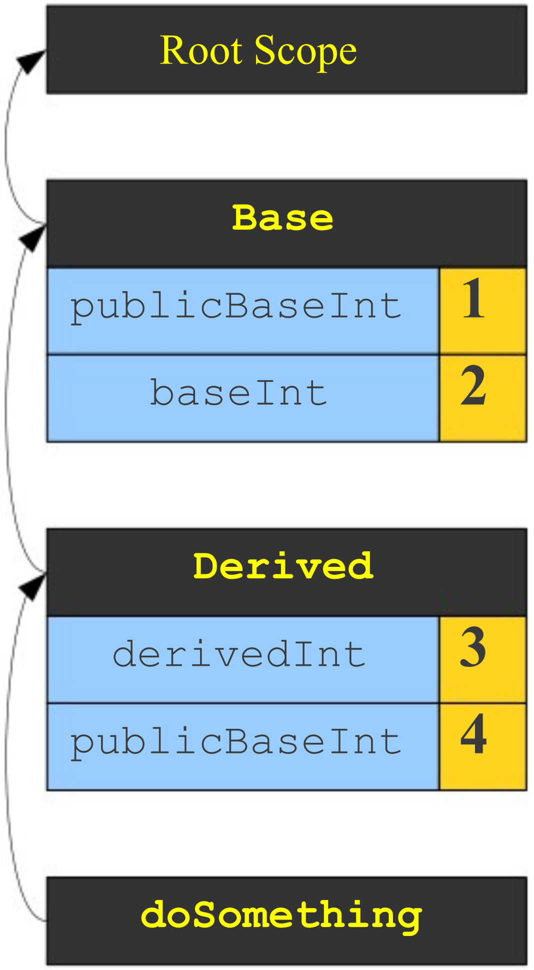
```
>
```



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);
    }
}
```

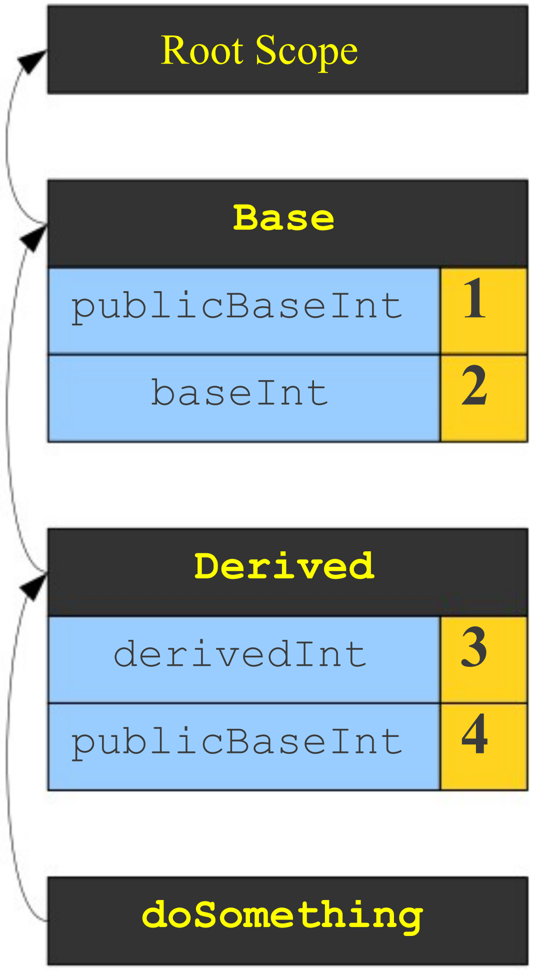
```
>
```



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);
    }
}
```

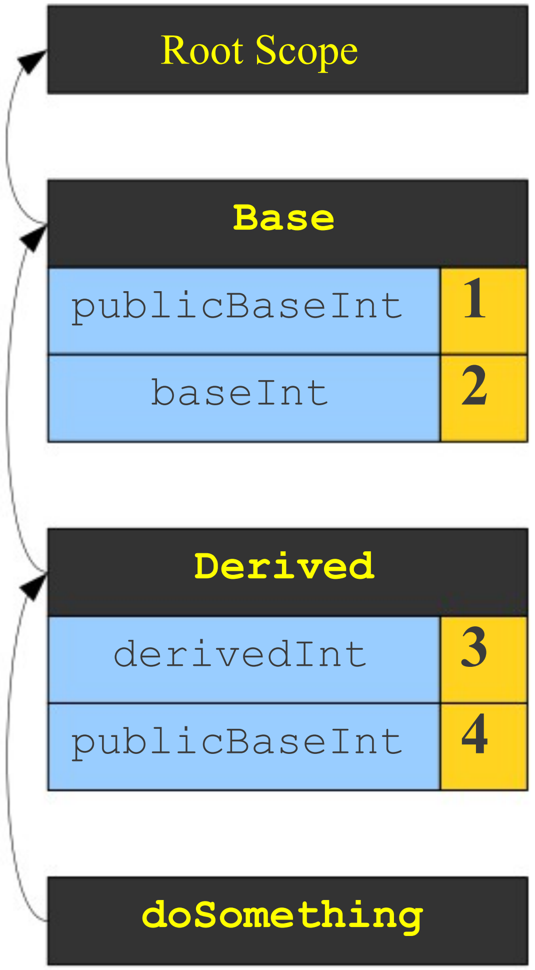
```
> 4
```



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);
    }
}
```

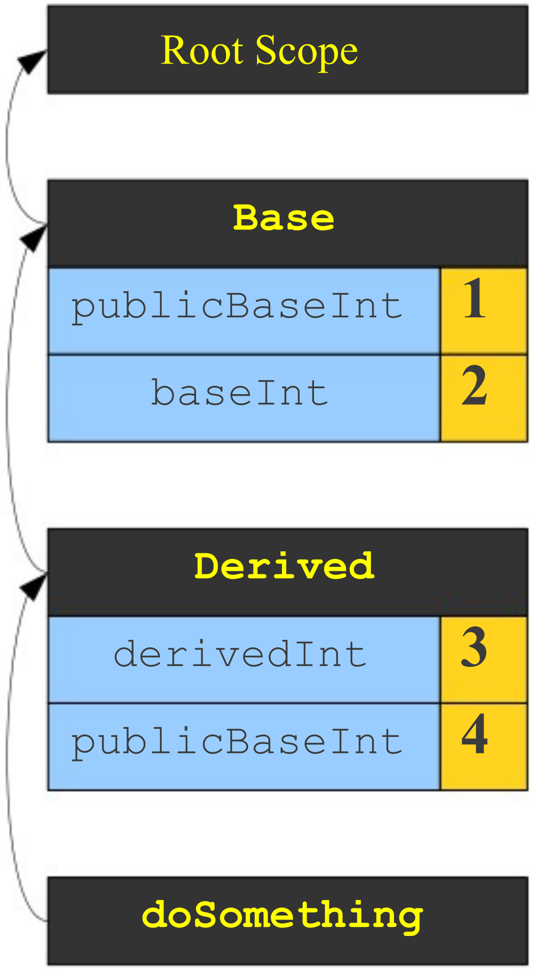
```
> 4
```



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);
    }
}
```

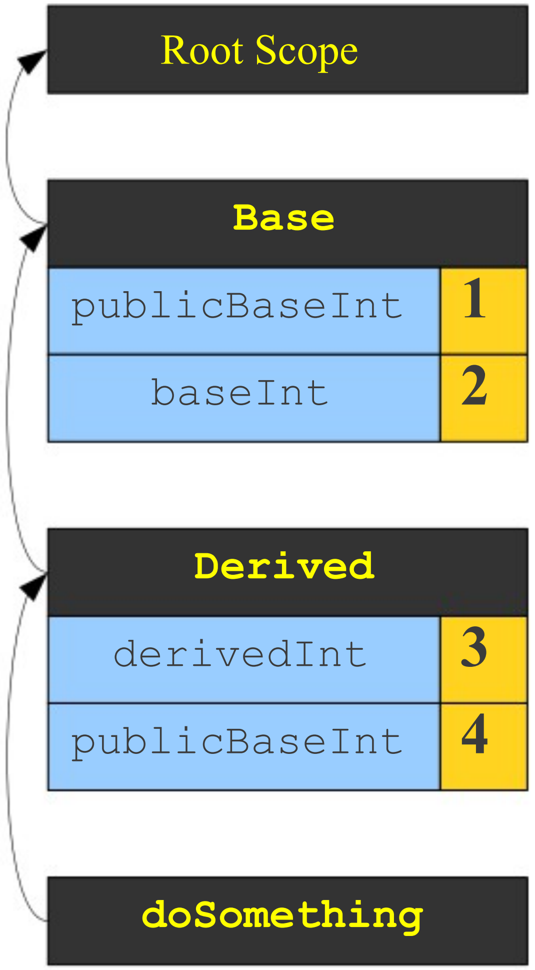
```
> 4
  2
```



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);
    }
}
```

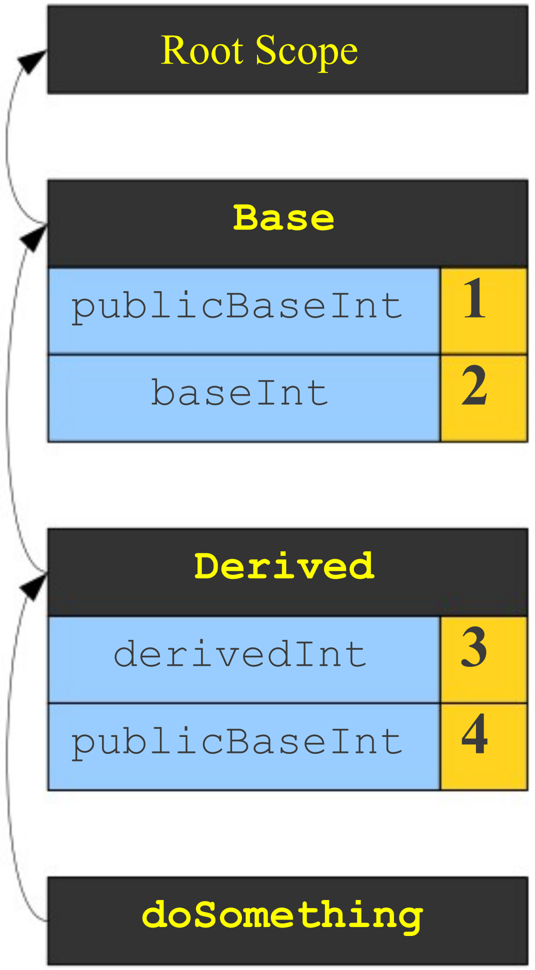
```
> 4
  2
```



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);
    }
}
```

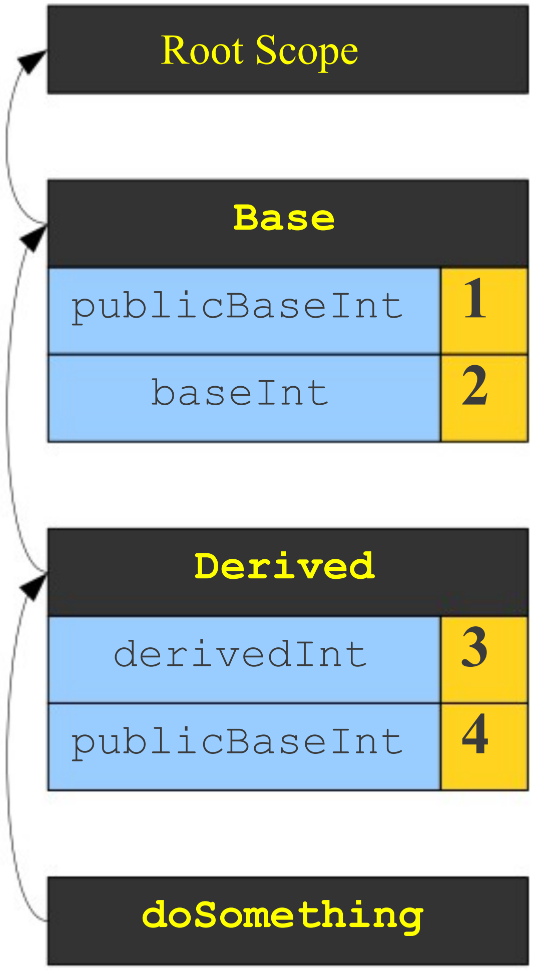
```
> 4
  2
  3
```



Scoping with Inheritance

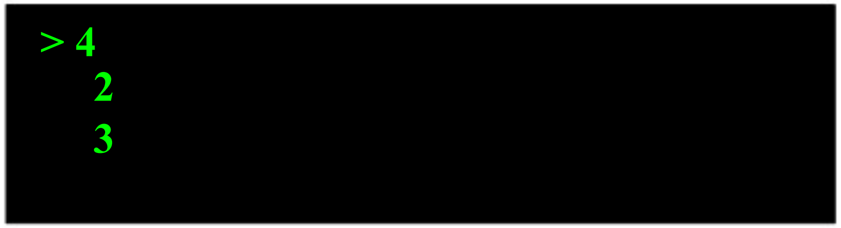
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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
```

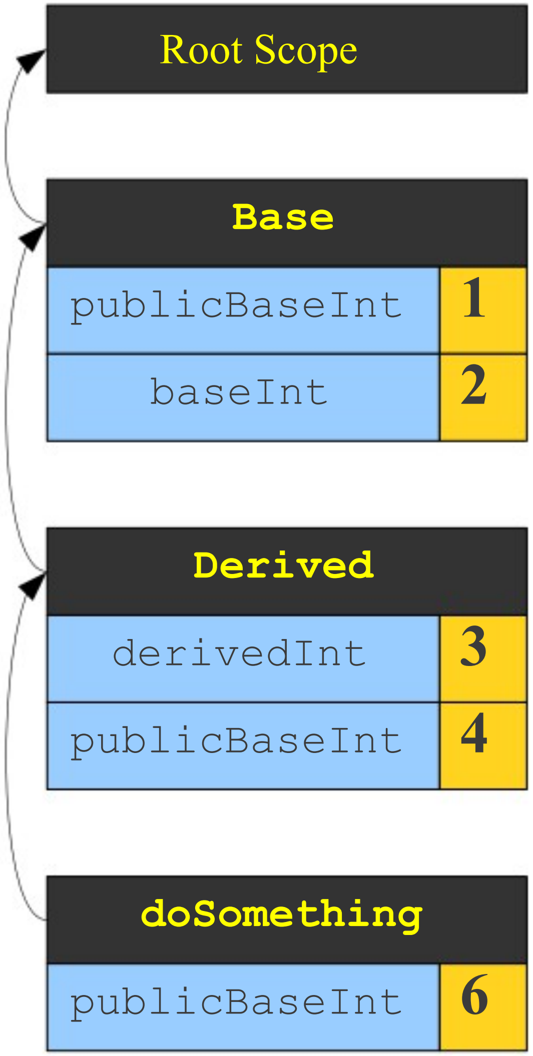


Scoping with Inheritance

```
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);
    }
}
```



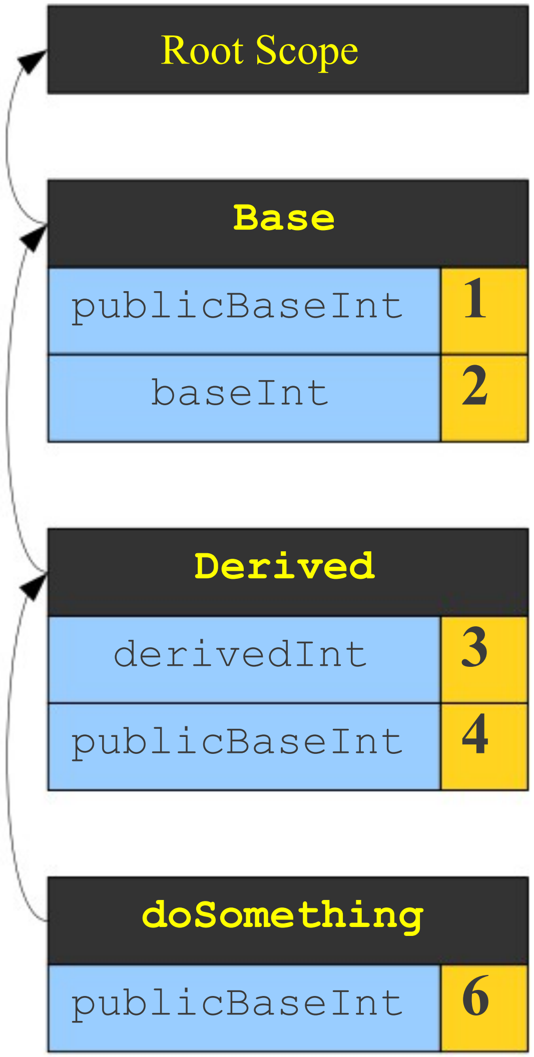
```
> 4
  2
  3
```



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);
    }
}
```

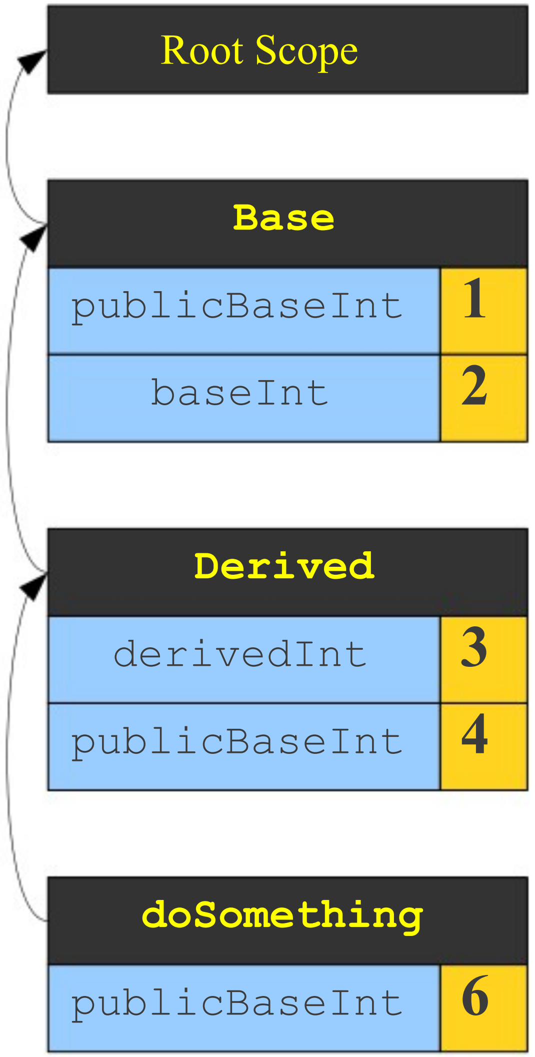
```
> 4
  2
  3
```



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);
    }
}
```

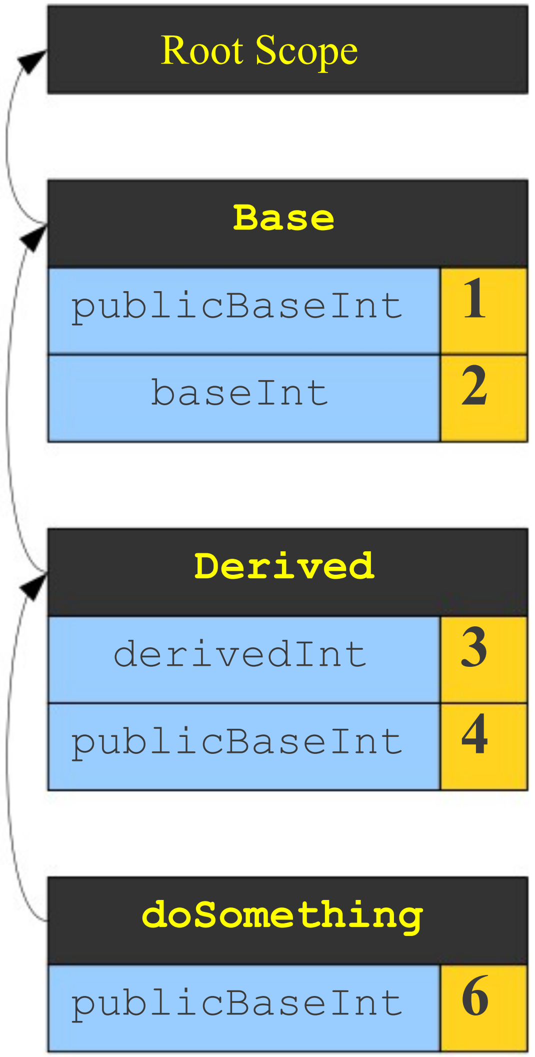
```
> 4
  2
  3
  6
```



Scoping with Inheritance

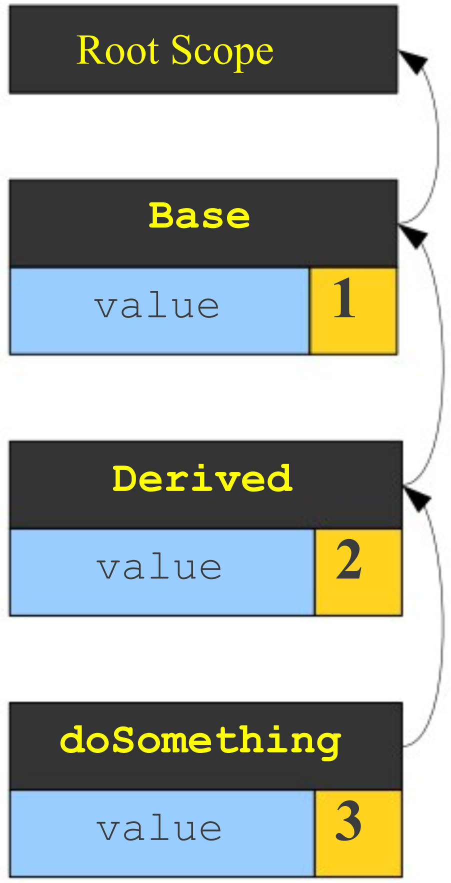
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

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
```



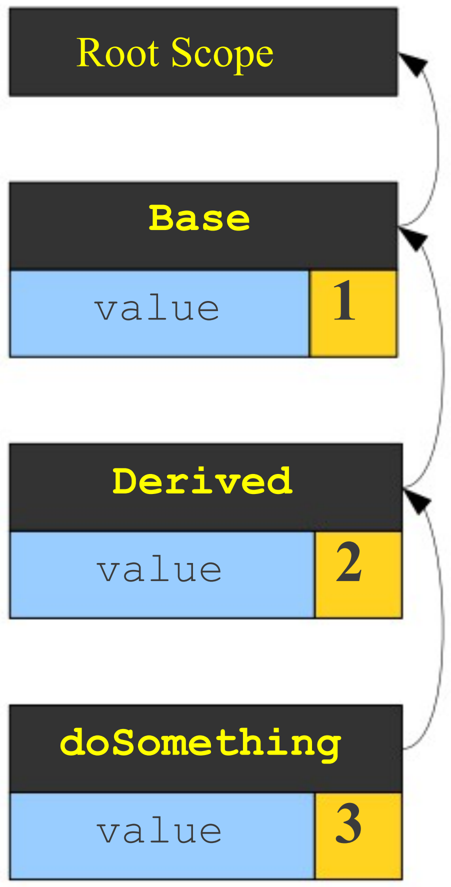
Explicit Disambiguation

```
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);
    }
}
```



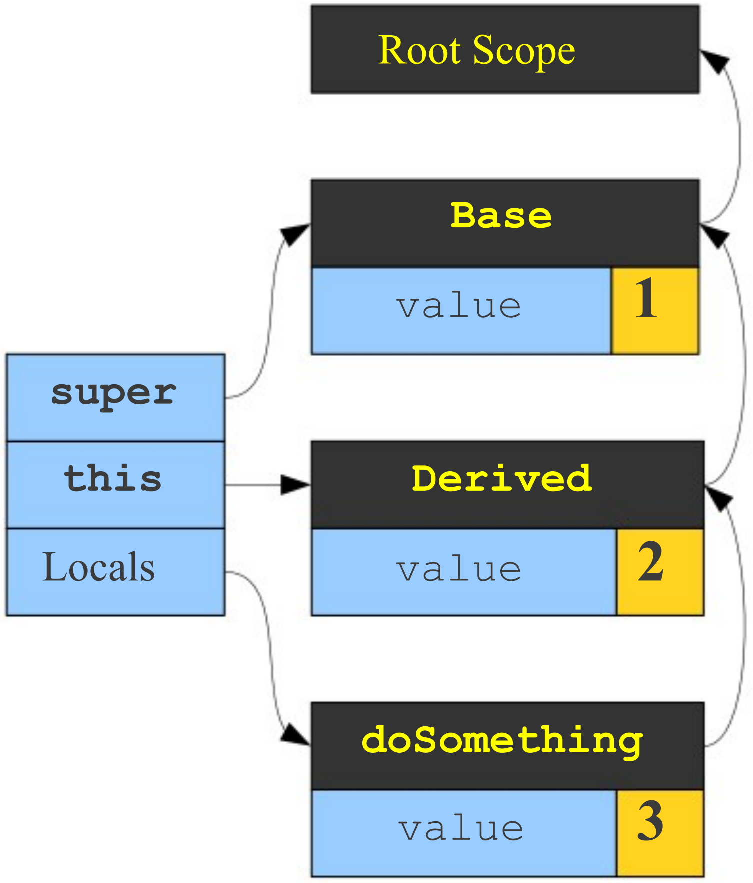
Explicit Disambiguation

```
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);
    }
}
```



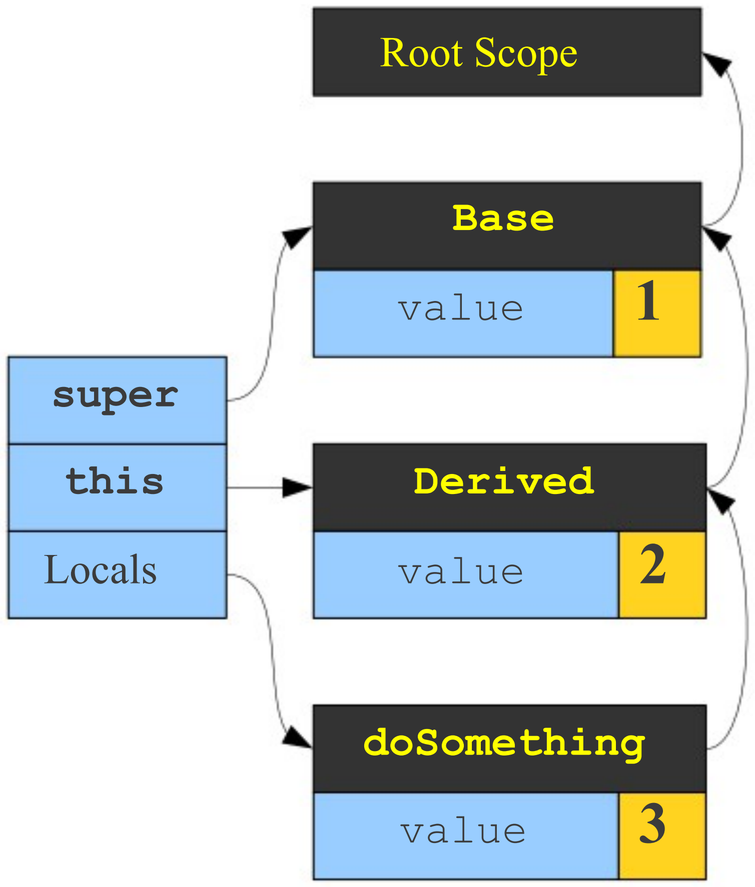
Explicit Disambiguation

```
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

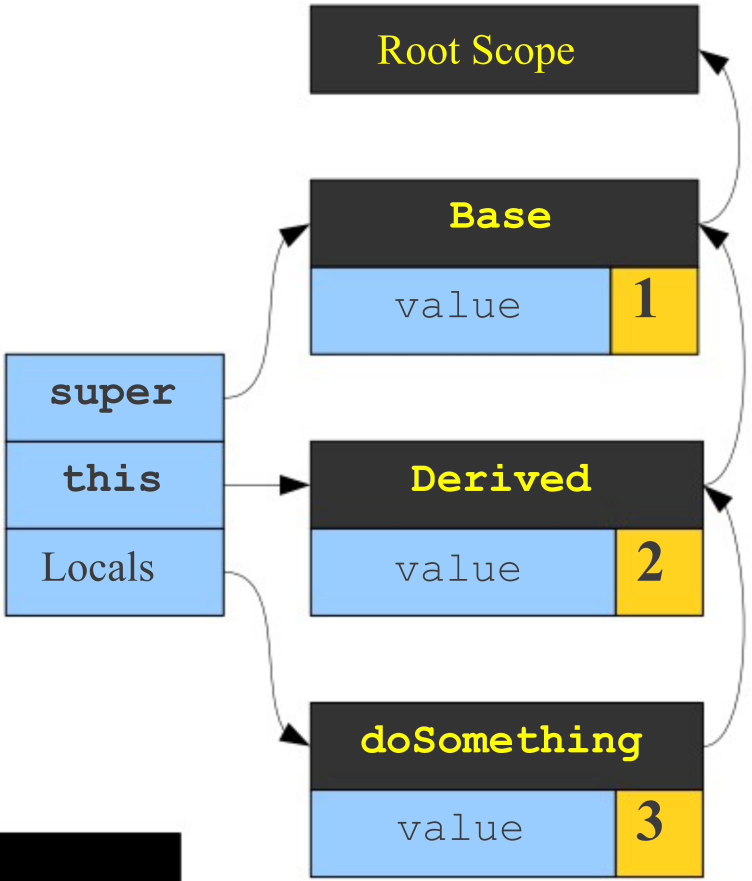
```
public class Base {
    public int value = 1;
}

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);
    }
}
```

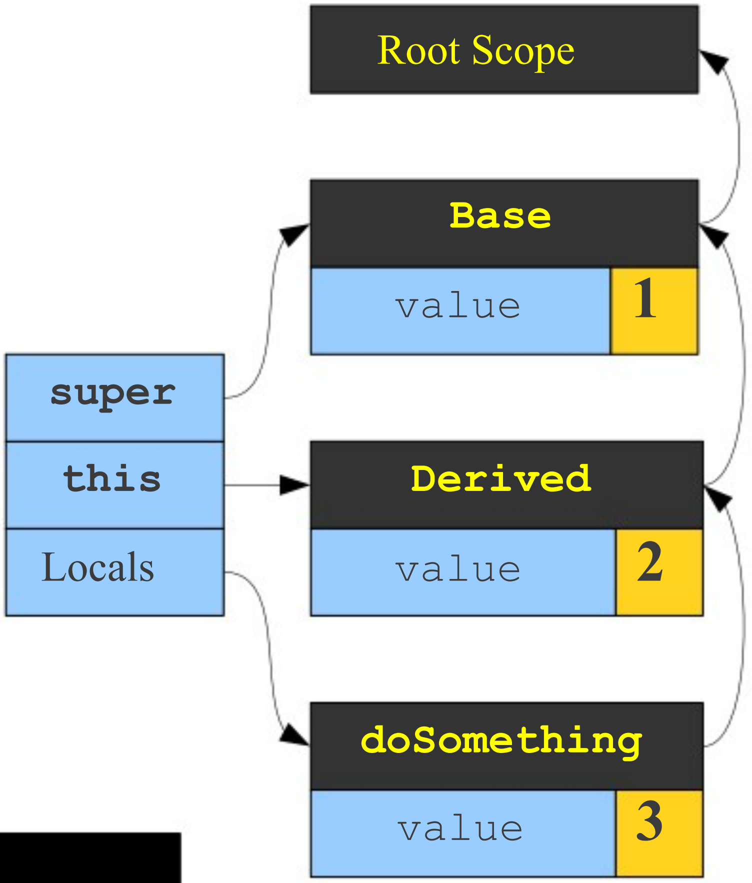
```
>
```



Explicit Disambiguation

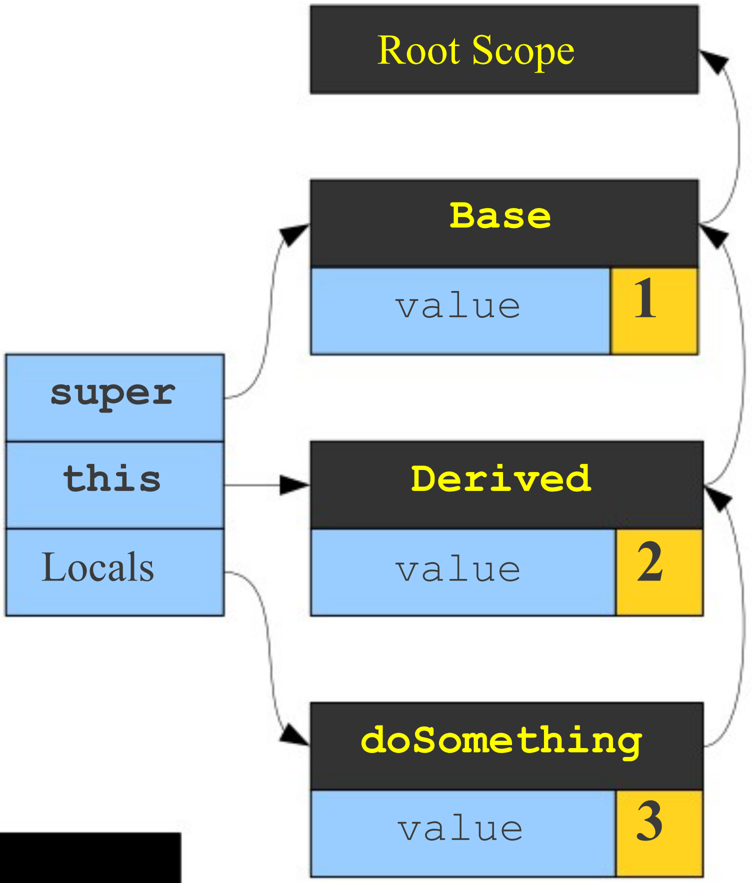
```
public class Base {
    public int value = 1;
}

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

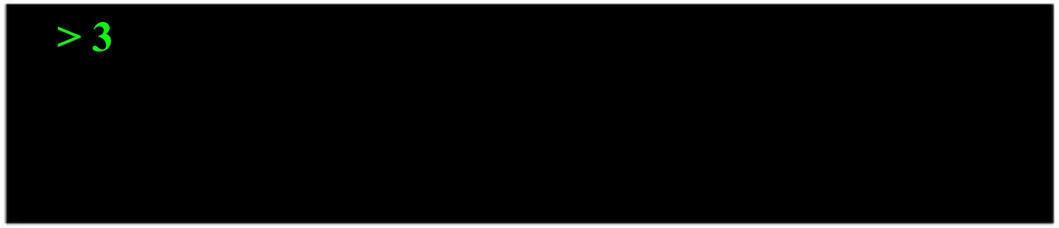
```
public class Base {
    public int value = 1;
}

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);
    }
}
```



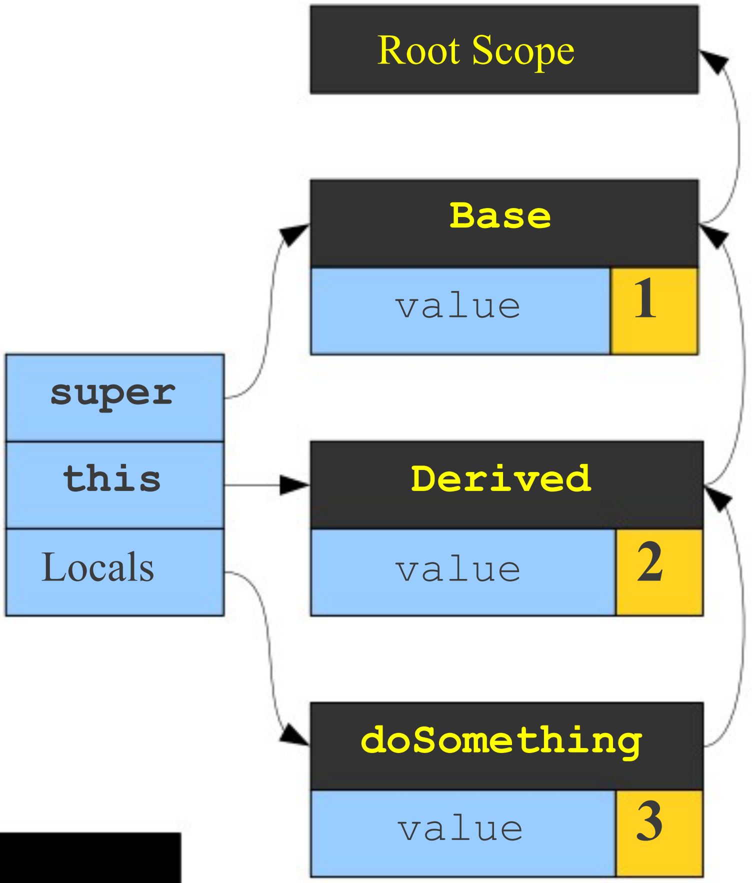
```
> 3
```

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);
    }
}
```

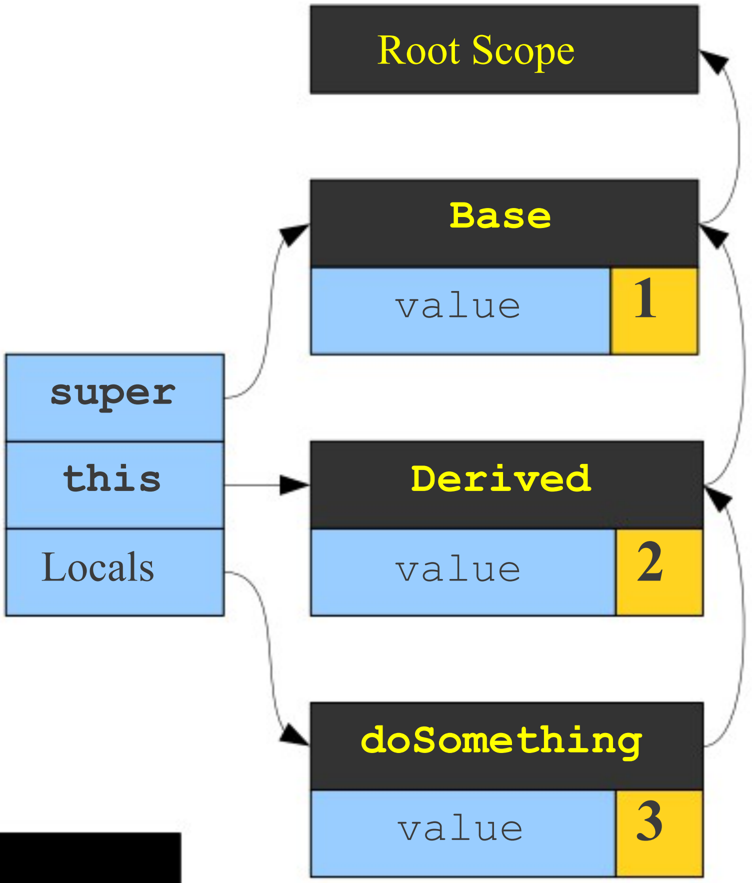


```
> 3
```



Explicit Disambiguation

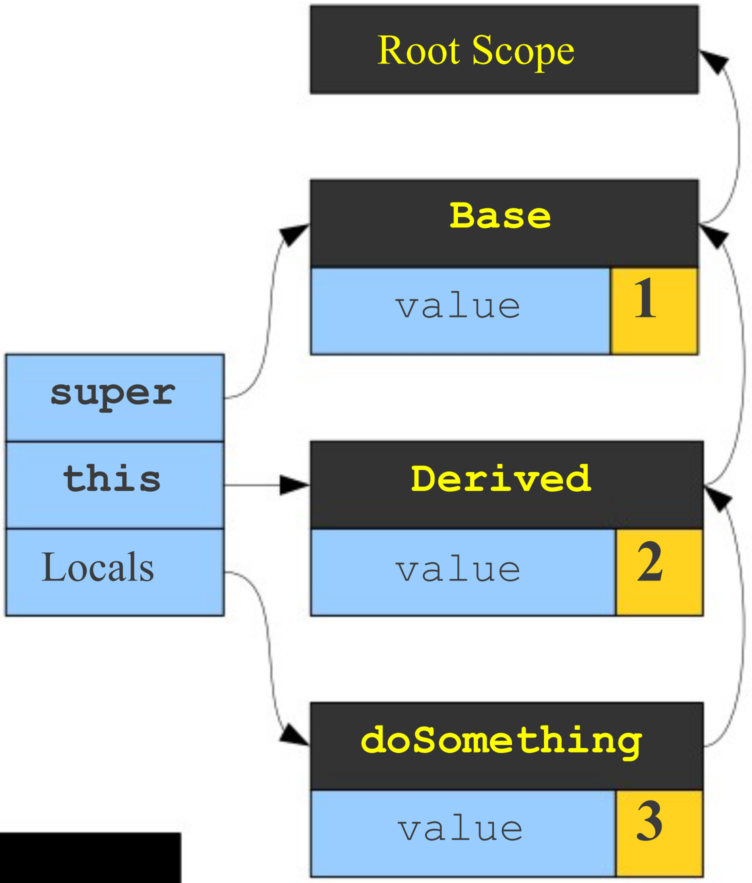
```
public class Base {
    public int value = 1;
}

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);
    }
}
```

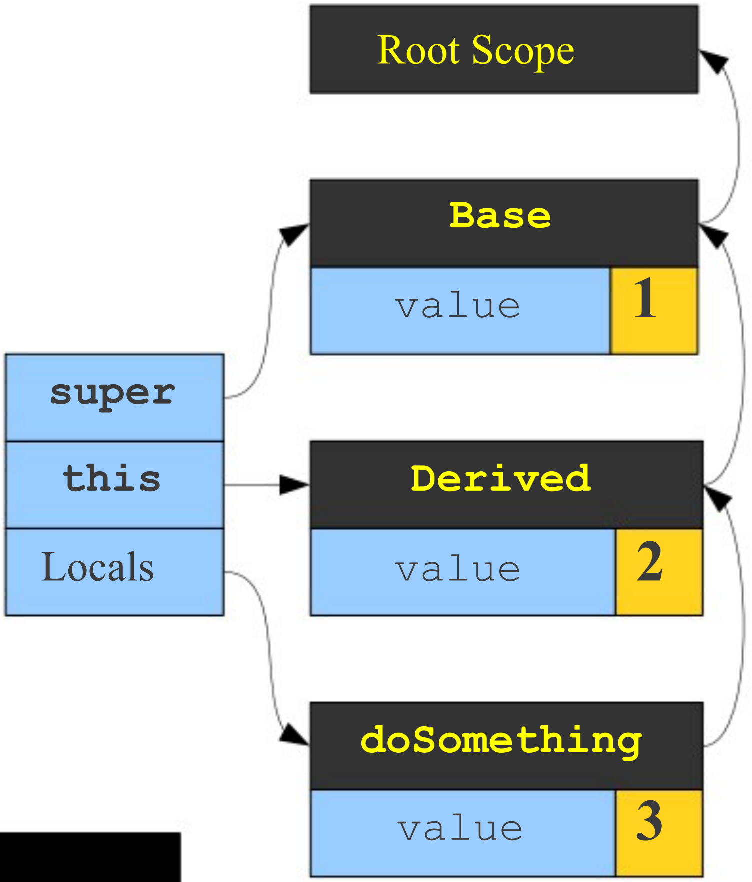


```
> 3
  2
```

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);
    }
}
```

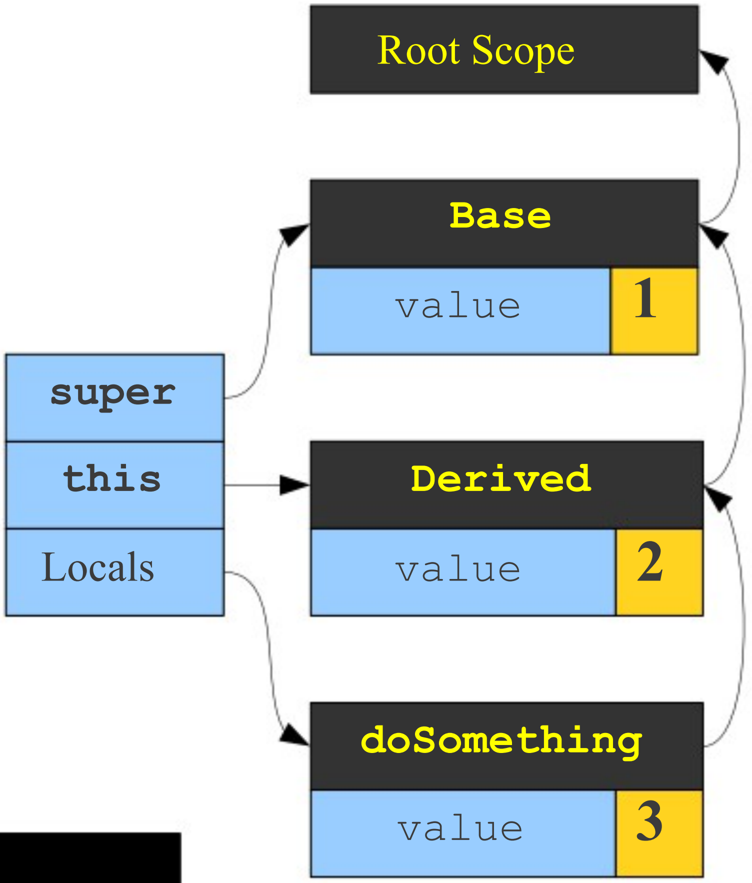
```
> 3
  2
  1
```



Explicit Disambiguation

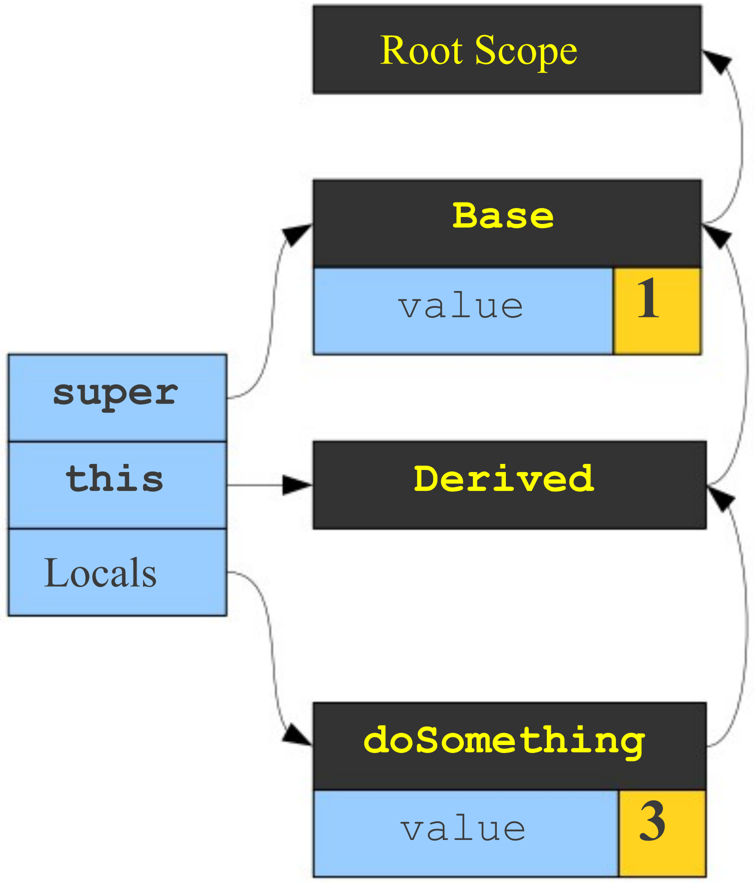
```
public class Base {
    public int value = 1;
}

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
```



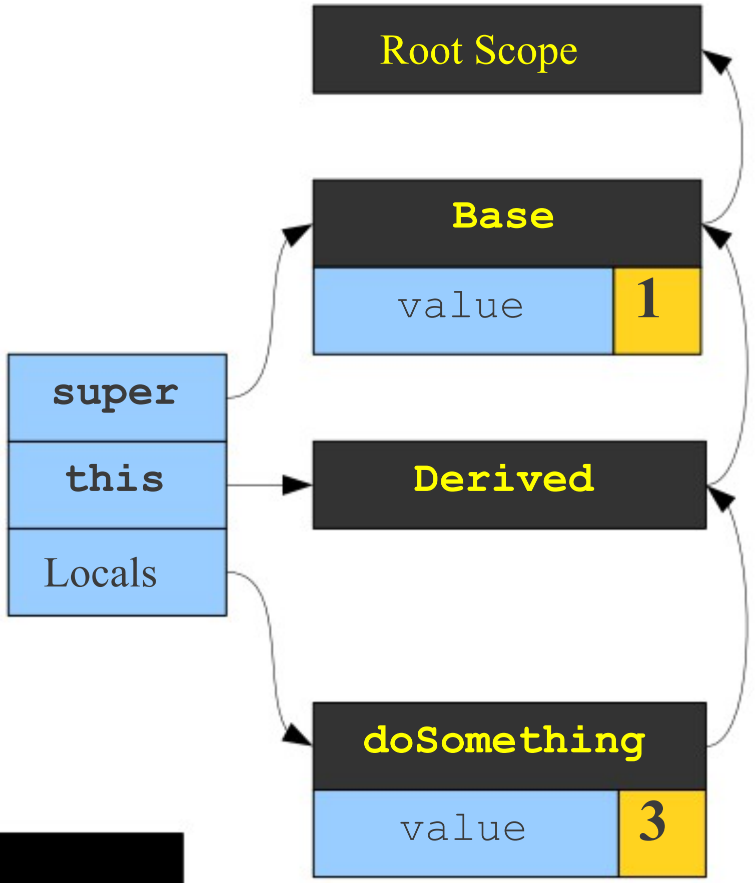
Explicit Disambiguation – no value Declared in Derived

```
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

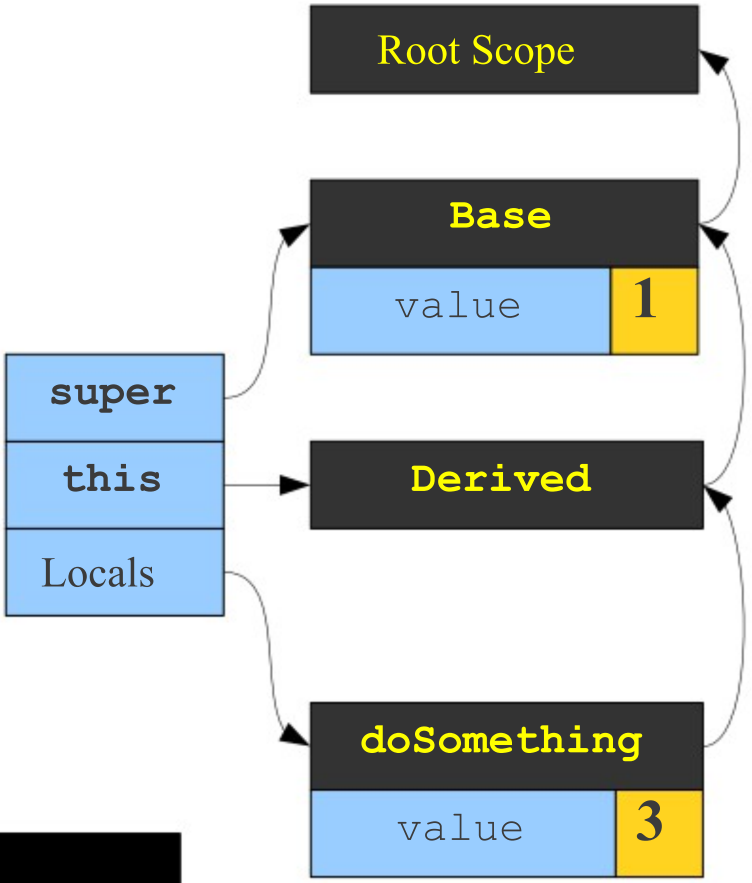
```
public class Base {
    public int value = 1;
}

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);
    }
}
```

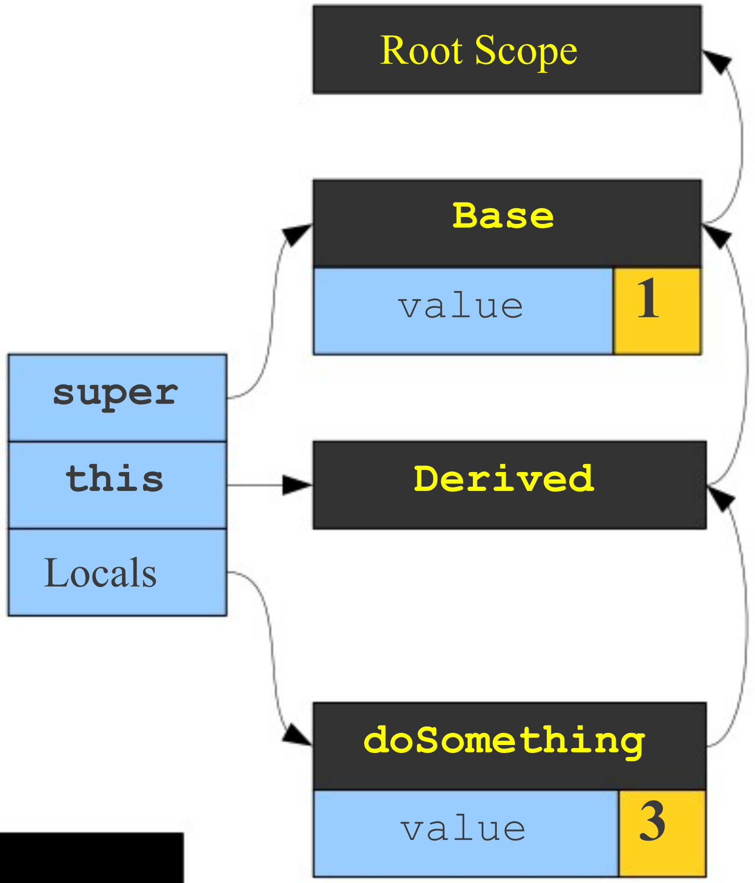


```
>
```

Explicit Disambiguation

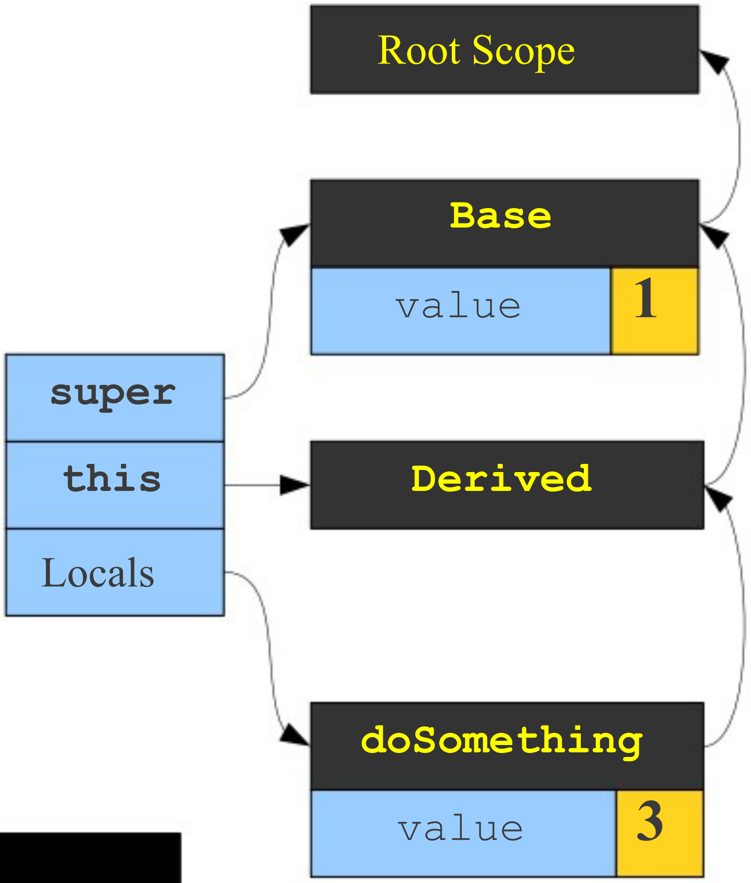
```
public class Base {
    public int value = 1;
}

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

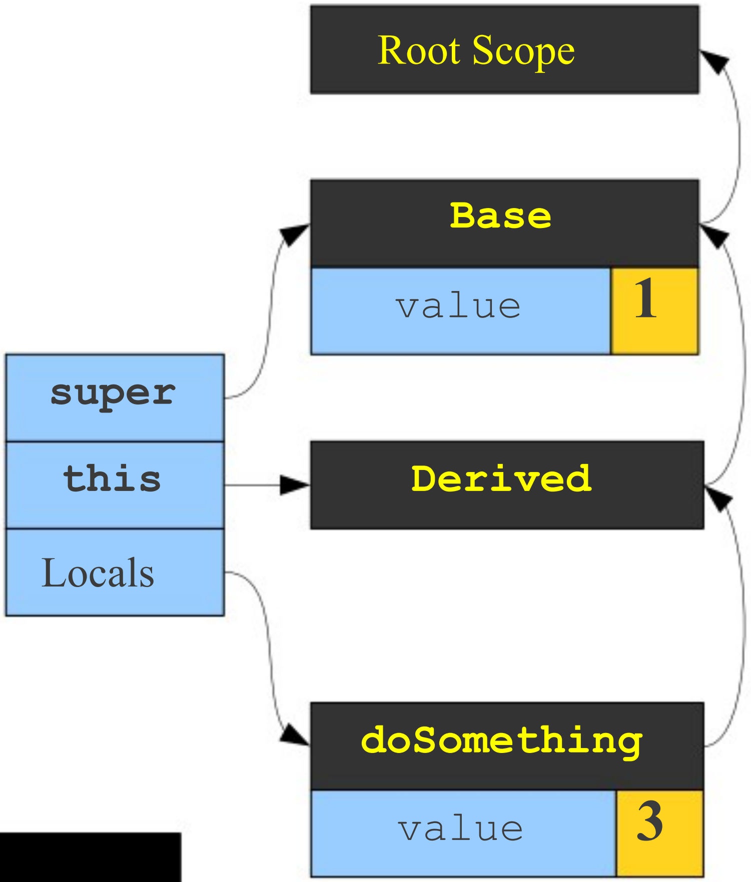
```
public class Base {
    public int value = 1;
}

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

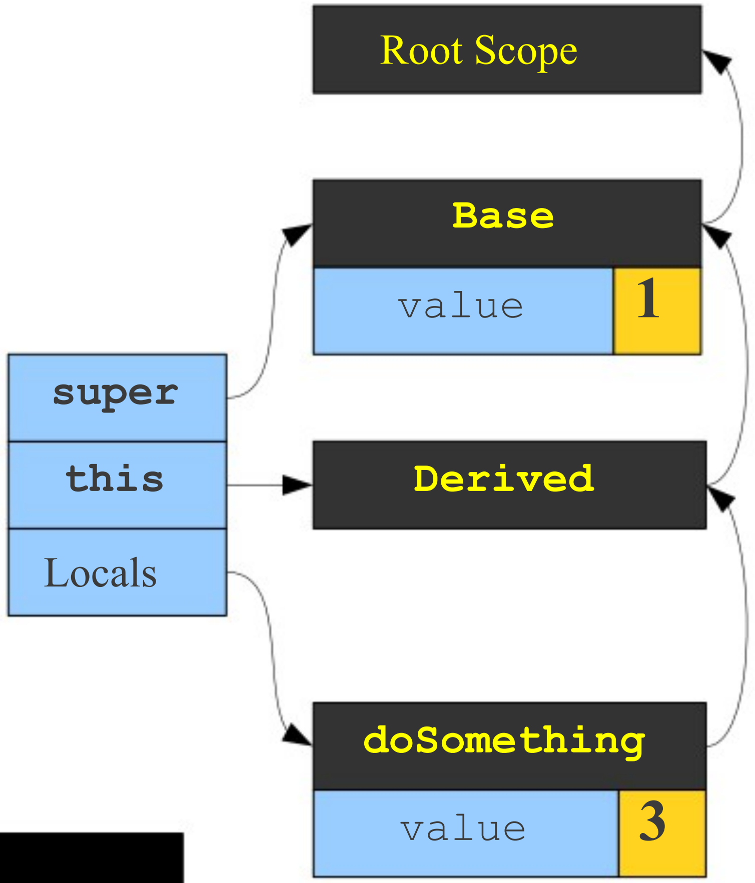
```
public class Base {
    public int value = 1;
}

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);
    }
}
```

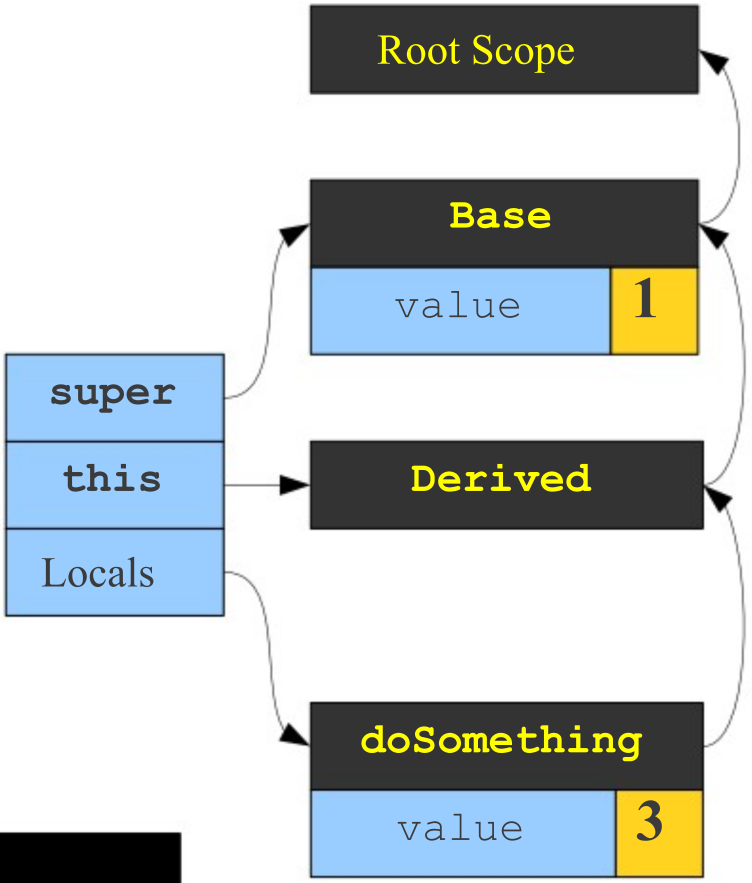


```
> 3
  1
```

Explicit Disambiguation

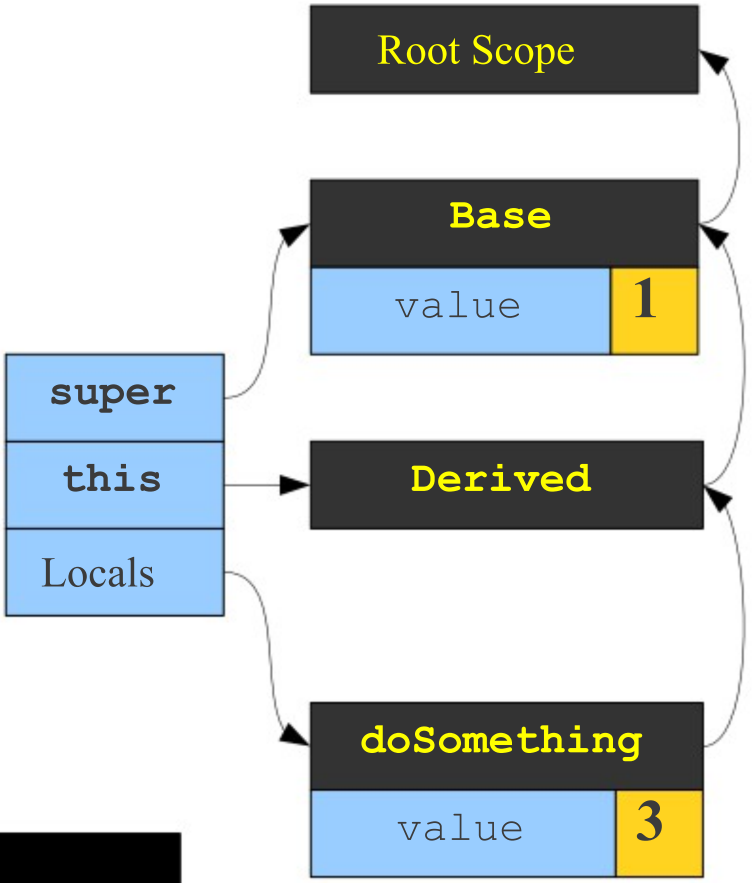
```
public class Base {
    public int value = 1;
}

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);
    }
}
```

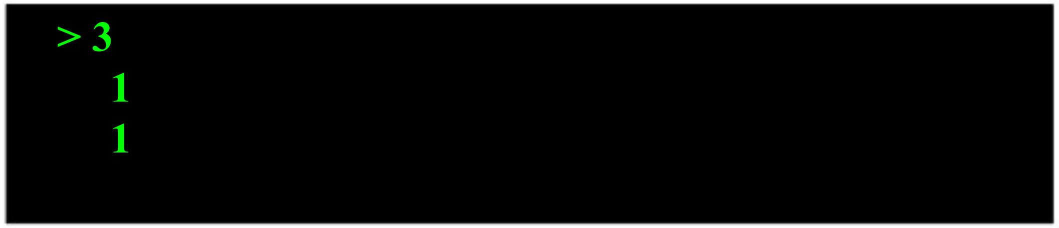
```
> 3
  1
  1
```



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
```

Summary

- Context-sensitive analysis is…
- Intermediate representations are needed because…
- A symbol table is used for…
- Scope checking
  - definition
  - how do you do it?

Where we are…

- ~~Admin and overview~~
- ~~Lexical analysis~~
- ~~Parsing~~
- **Semantic analysis**
- Machine-independent optimisation
- Code generation
- Hardware architectures
- Machine-dependent optimisation
- Review