Compiler Construction
Winter 2020

Recitation 3:
Symbol Tables

Yotam Feldman

Based on slides by Guy Golan-Gueta

Simple Example.java

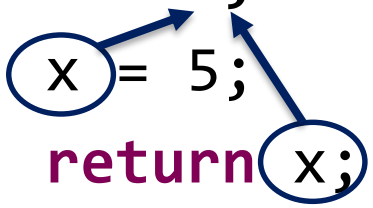
```
class Main {
    public static void main(String[] a) {
        System.out.println(new Simple().Start(1, 1));
    }
}

class Trivial {
        int f;
}
```

```
class Simple extends Trivial {
        public int Start(int a, int b) {
                int x;
                int y;

                x = a;
                y = b + 3;

                if (true) {
                        f = 0;
                } else {
                        f = 1;
                }

                return x + y + f;
        }
}
```

Symbol Resolution

Resolving Variables: Local Variables

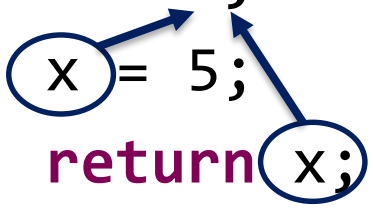
```
class A {
      public int bar() {
            int x;
            x = 5;
            return x;
      }
}
```



1. Defined
2. Type
3. Visibility
   (later, where to store it)

Resolving Variables: Local Variables

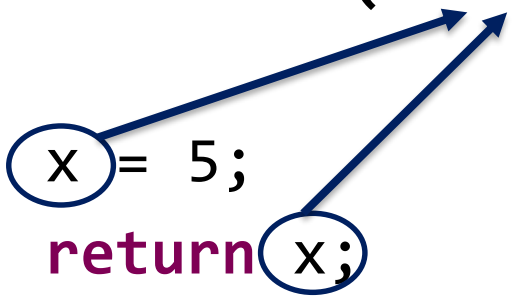
```
class A {
      public int bar1() {
            int x;
            x = 5;
            return x;
      }
      public int bar2() {

            x = 9;
            return x;
      }
}
```



error

Resolving Variables: Parameters

```
class A {
      public int bar(int x) {
            x = 5;
            return x;
      }
}
```



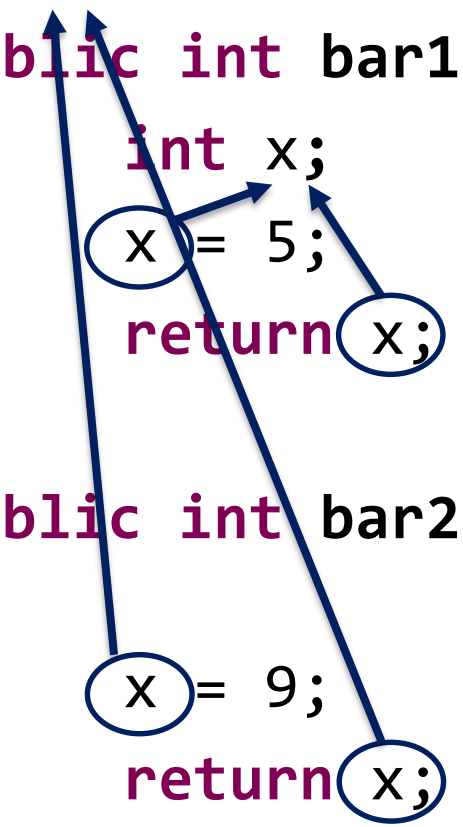
Resolving Variables: Parameters

```
class A {
      public int bar(int x) {
            int x;
            x = 5;
            return x;
      }
}
```

error

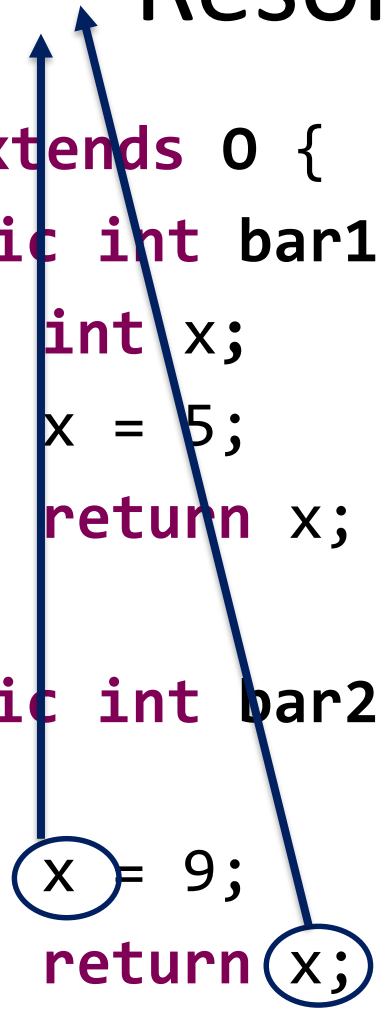
Resolving Variables: Fields

```
class A {
        int x;
        public int bar1() {
                int x;
                x = 5;
                return x;
        }
        public int bar2() {

                x = 9;
                return x;
        }
}
```



Resolving Variables: Fields

```
class O {
        int x;
}
class A extends O {
      public int bar1() {
            int x;
            x = 5;
            return x;
      }
      public int bar2() {

            x = 9;
            return x;
      }
}
```



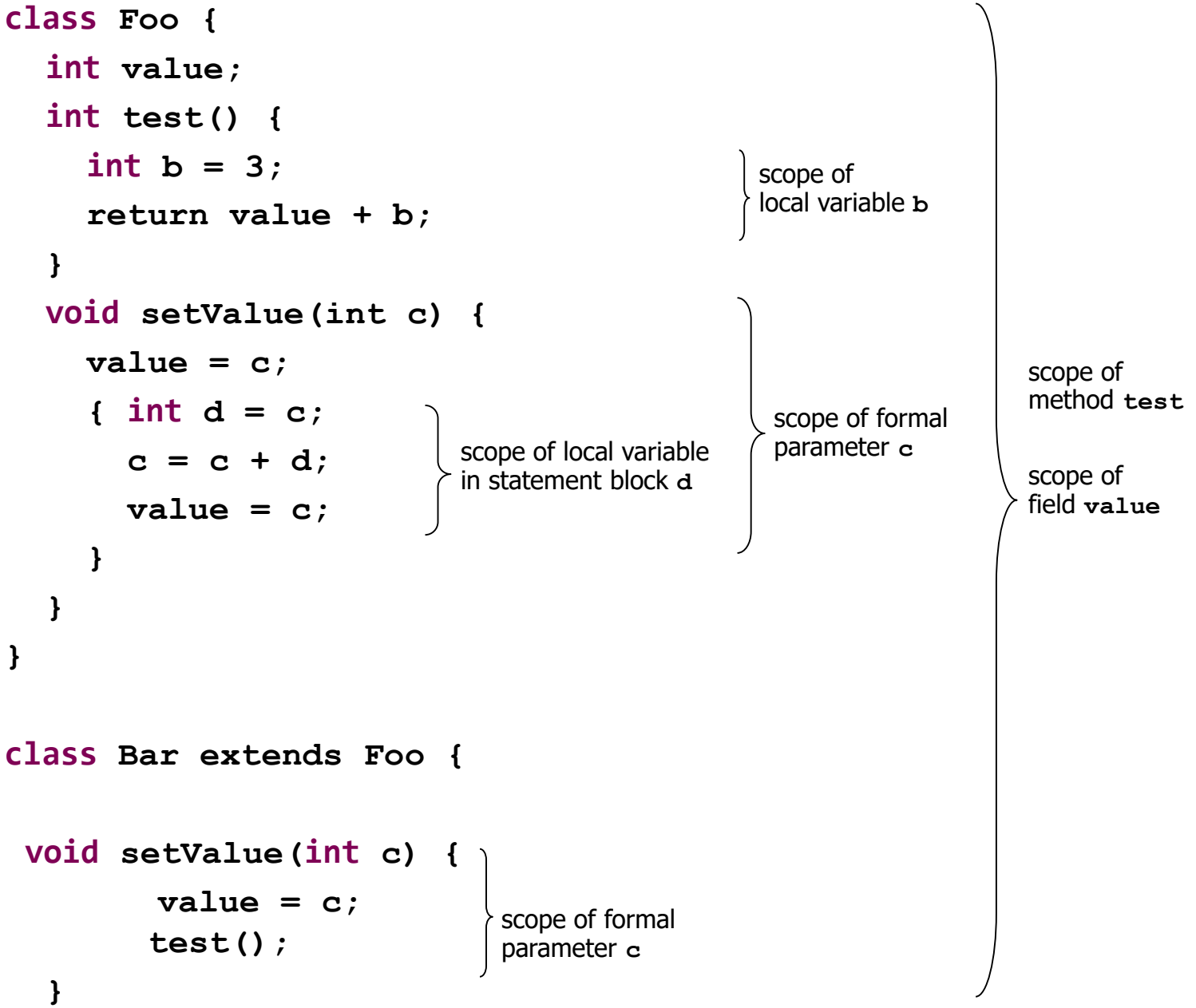
Resolving Variable Names

- Basic building block
- Ex 1: renaming occurrences of a variable
- Ex 2: translate variables to memory locations / registers
- Ex 3: all uses are valid

Scope

- Scope of identifier
  - portion of program where identifier can be referred to

- Scope
  - Statement block
  - Method body
  - Class body
  - Module / package / file
  - Whole program (multiple modules)


class Foo {
int value;
int test() {
int b = 3;
return value + b;
}
void setValue(int c) {
value = c;
{ int d = c;
c = c + d;
value = c;
}
}
}
class Bar extends Foo {
void setValue(int c) {
value = c;
test();
}
}
scope of
local variable b
scope of formal
parameter c
scope of local variable
in statement block d
scope of
method test
scope of
field value
scope of formal
parameter c

Scope Hierarchy in MiniJava

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

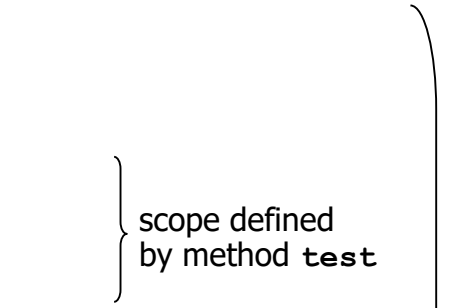
Symbol Table

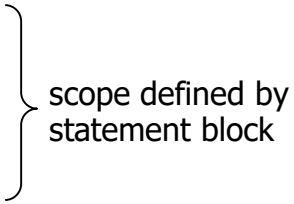
- An environment that stores information about identifiers
- A data structure that captures scope information
- One symbol table for each scope

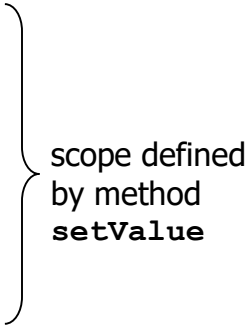
Symbol	Kind	Decl	Properties
value	field	int	…
test	method	-> int	…
setValue	method	int -> void	…

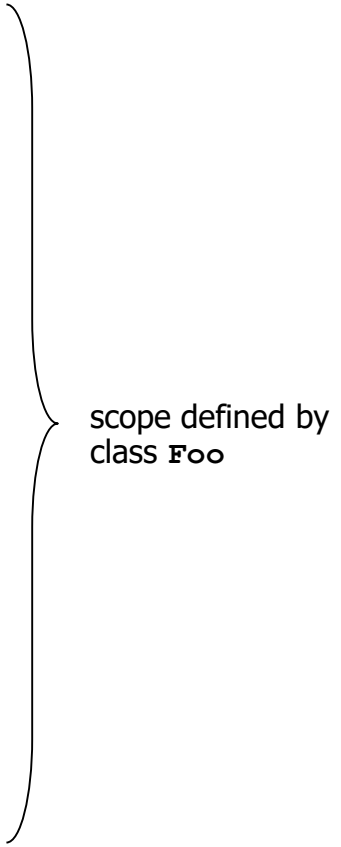
Symbol Table Example

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



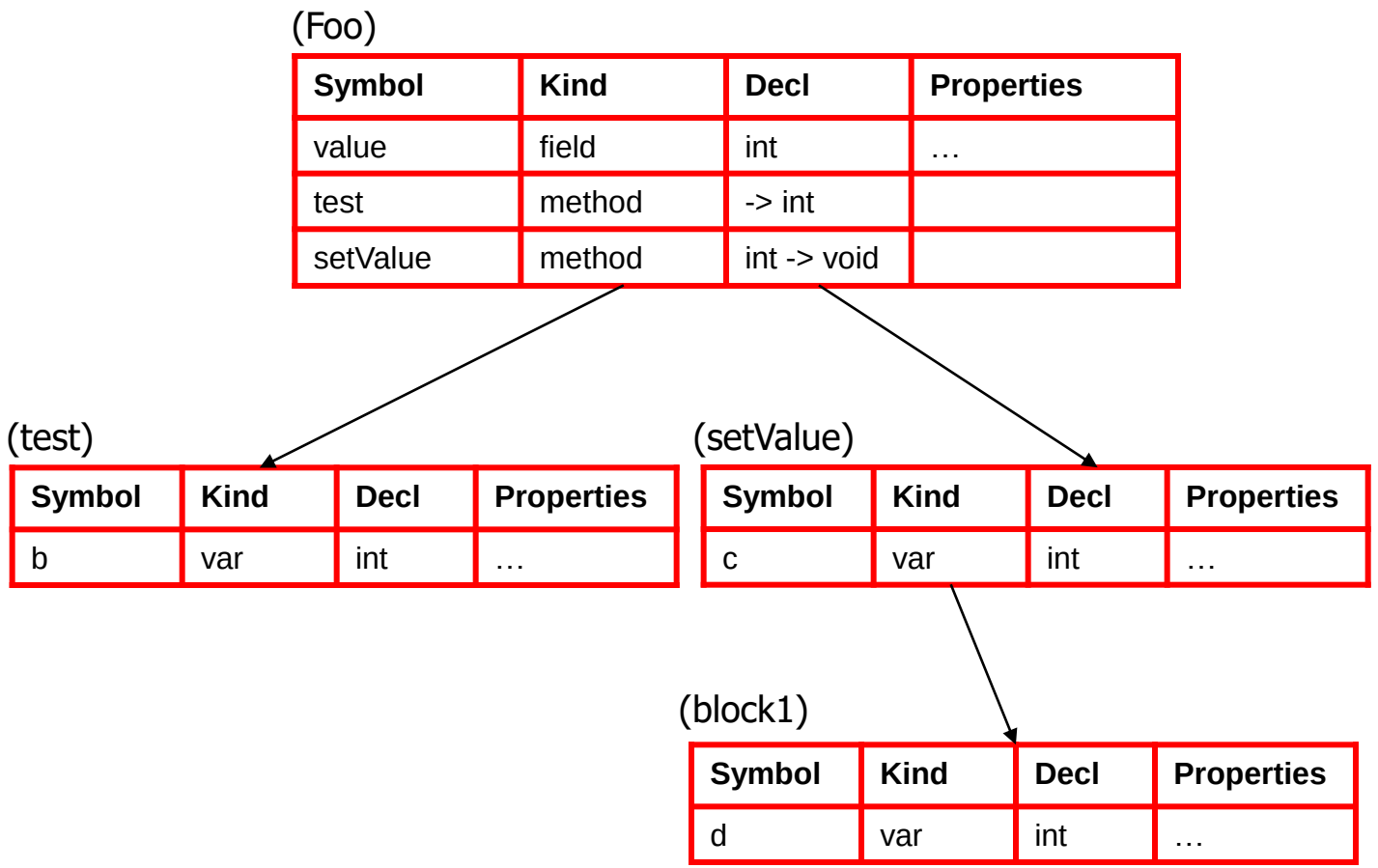




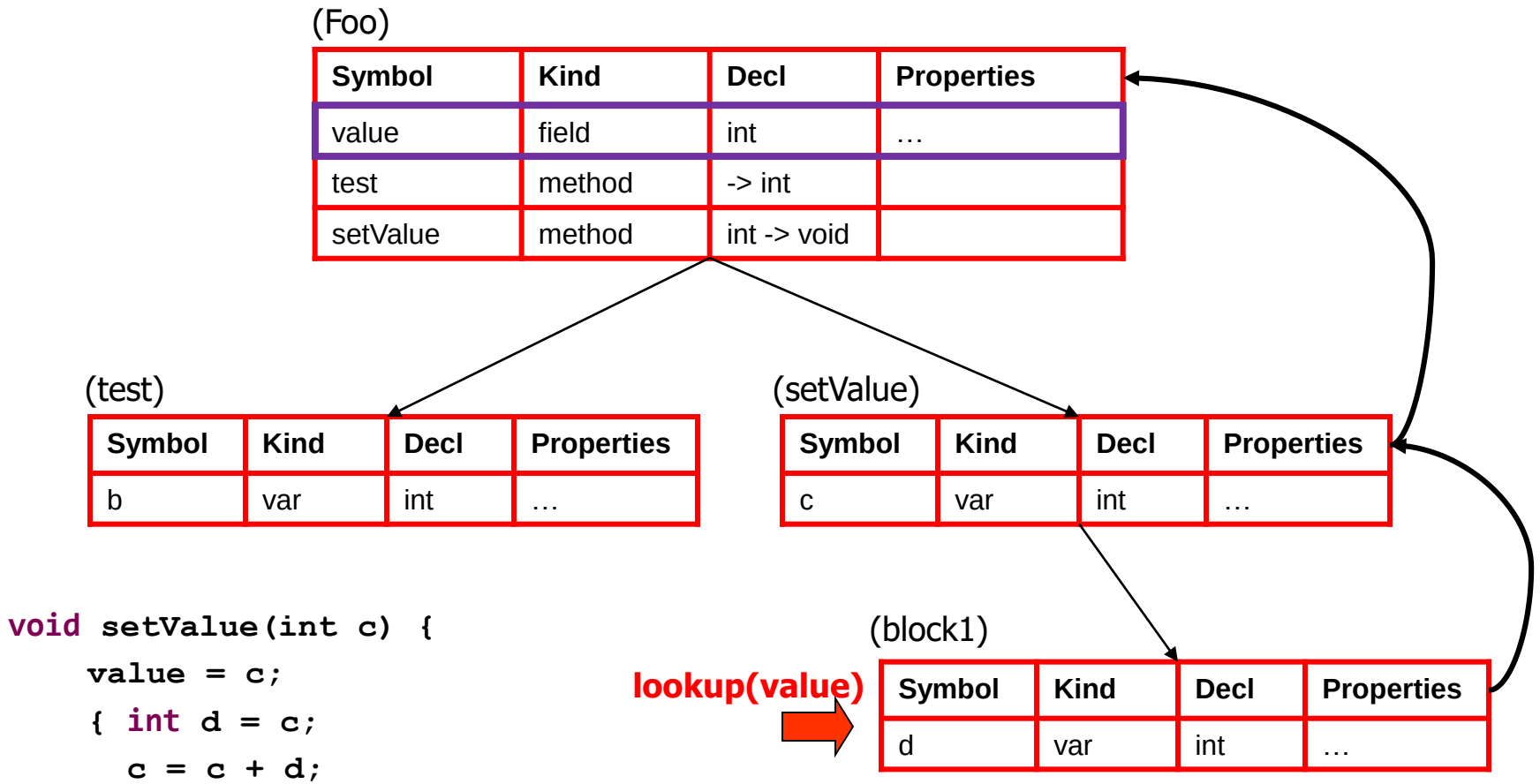


Symbol Table Example

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

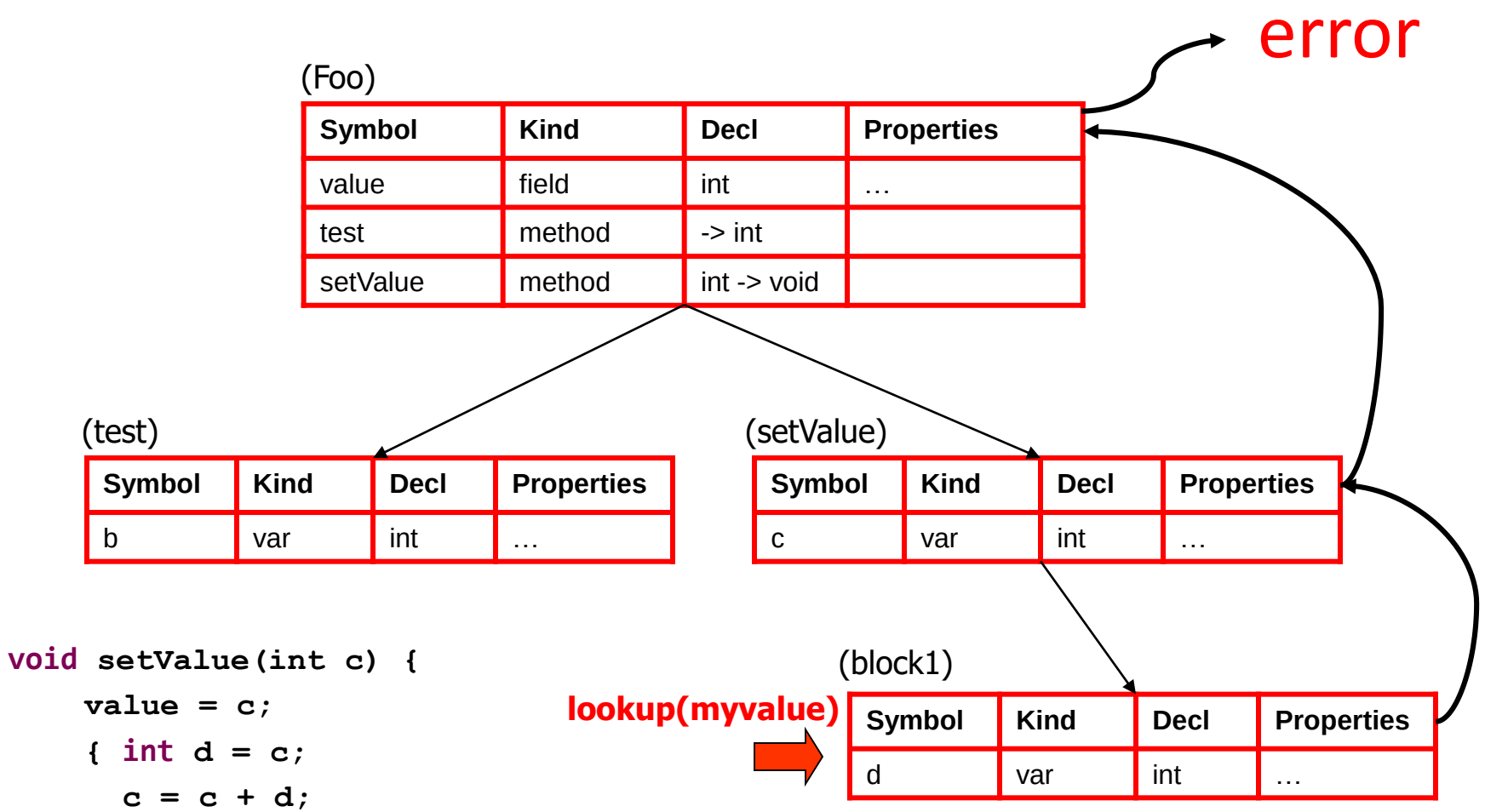


Resolving Variable Names




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

Resolving Variable Names




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

Symbol Table Implementation

```
public class SymbolTable {
  private Map<String,Symbol> entries;
  private SymbolTable parentSymbolTable;
…
}

public class Symbol {
  private String id;
  …
}
```

My personal preferences:

- Different functions for lookup for variables and for methods
- Mapping from the symbol to its AST declaration

Accessing the Symbol Table

- Need the symbol table of the current scope
- Can either
  - Add a field to each AST node,
  - Construct a global map from AST nodes to their symbol table, or
  - Keep track of it during the traversal (in the visitor)

Accessing the Symbol Table: Adding a Field

```
public abstract class AstNode {
  public Integer lineNumber;

  /** reference to symbol table of enclosing scope **/
  private SymbolTable enclosingScope;

  /** returns symbol table of enclosing scope **/
  public SymbolTable enclosingScope() {…}
}
```

Multiple Passes

- Building visitor
  - A propagating visitor
  - Propagates reference to the symbol table of the current scope
- Whatever-it-is visitor
  - Resolve names using the symbol table

Summary

- Resolving variable names
  - Local variables
  - Formal parameters
  - Fields
- Scope
  - Scope nesting
- Symbol tables
- Ex 1