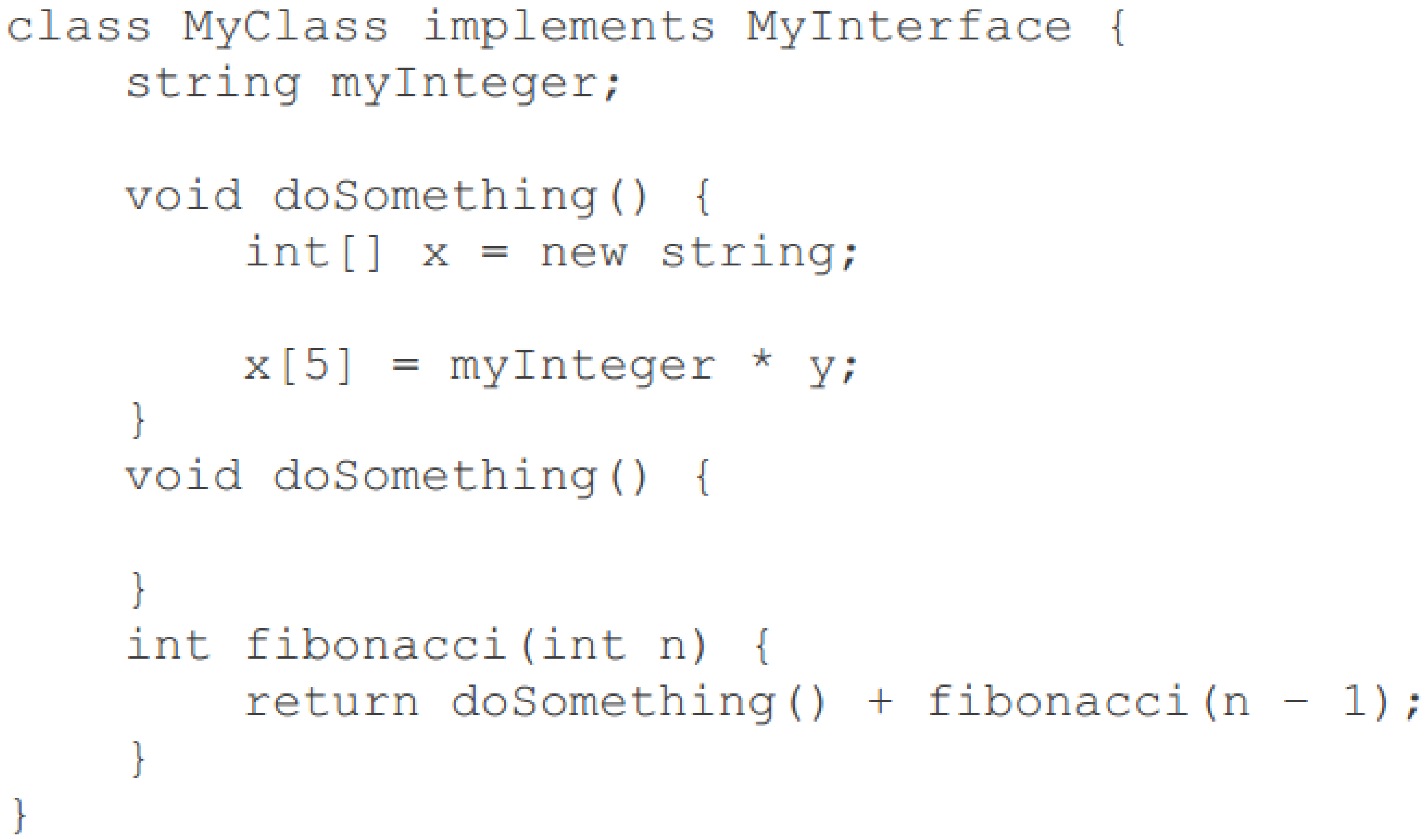
Semantic Analysis

Where We Are?

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

Consider the following program:

It is syntactically correct, but is it error free?

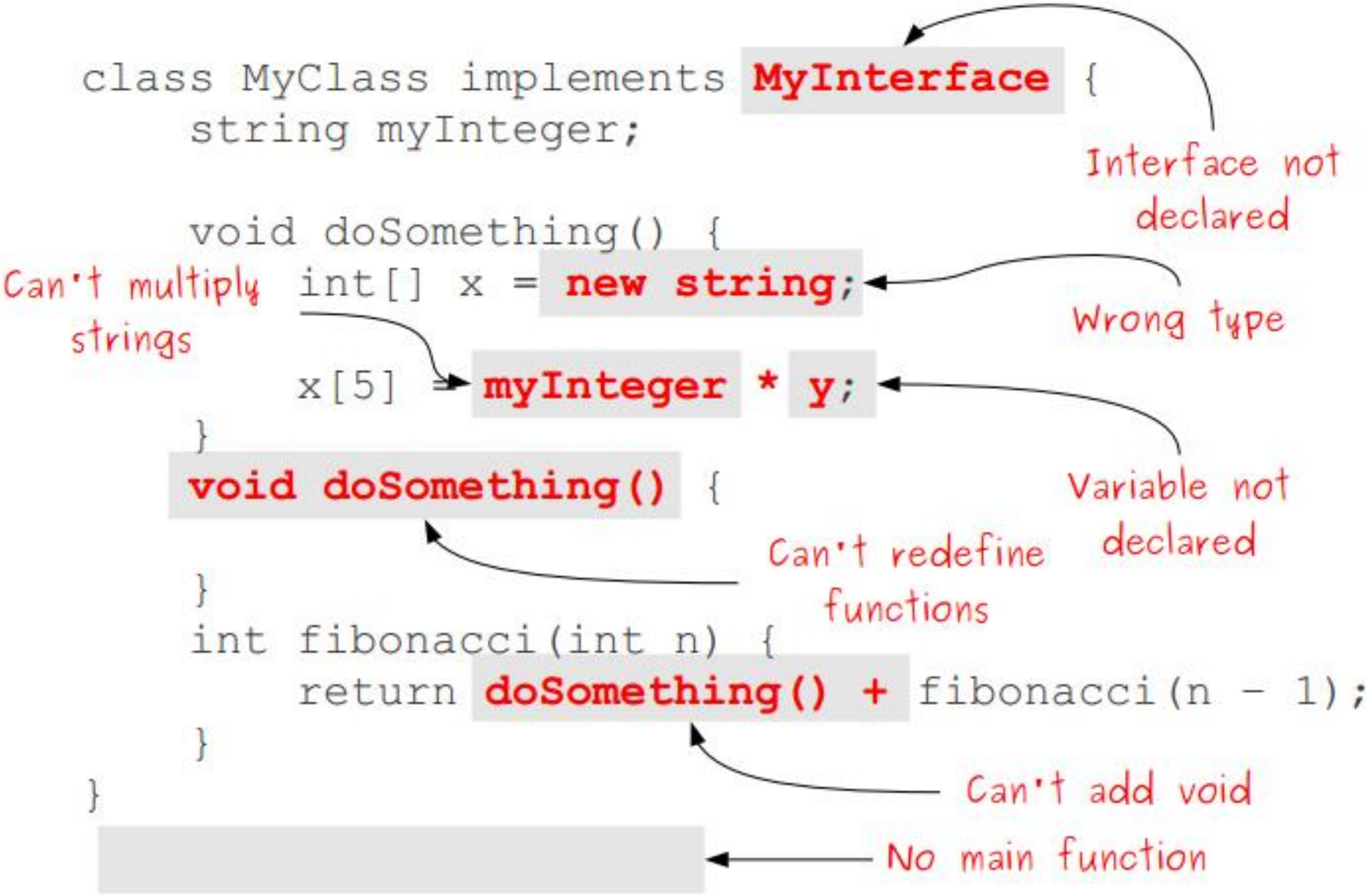


```
class MyClass implements MyInterface {
    string myInteger;

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

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

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



```
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 is our last line of defense.
- Parsing cannot catch all errors
- This is because CFG are **not expressive enough** to describe everything we are interested in in a language.
- **i.e., some language constructs are not context free**.

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**.

Implementing Semantic Analysis

- Attribute Grammars
  - Augment rules to do checking during parsing.
  - Approach suggested in the Compilers book.
  - Has its limitations; more on that later.
- Recursive AST Walk
  - Construct the AST, then use virtual functions and recursion to explore the tree.
  - The approach we'll take in this class.

Abstract Syntax Tree: AST

- Much of the semantic analysis can be expressed as a recursive descent of an AST.
- When we traverse an AST some operations are performed on a node before we process its children and some operations are performed after we process its children
  - Before: process an AST node n
  - Recurse: Process the children of n
  - After: Finish processing the AST node n
- This is called **Recursive Decent Traversal of a Tree**
  - Sometimes we process a node before its children, sometimes after, and sometimes both.
- When performing semantic analysis on a portion of the AST, we need to know **which identifiers are defined**

Types of Checks

- 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**?

Scope

- 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);
    }
}
```

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

- 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

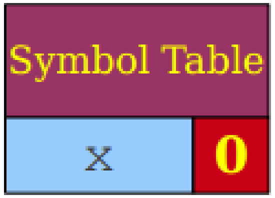
- A symbol table is a **mapping from a name to the thing that name refers to**.
- As we run our **semantic analysis, continuously update the symbol table** with information about what is in scope.
- Questions:
  - What does this look like in practice?
  - **What operations** need to be defined on it?
  - How do we **implement** it?

Symbol Tables: The Intuition

```
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);
  }
}
```

Symbol Tables: The Intuition

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



Symbol Table	
x	0

Symbol Tables: The Intuition

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

Symbol Table	
x	**0**
z	**1**

Symbol Tables: The Intuition

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

Symbol Table	
x	**0**
z	**1**
x	**2**
y	**2**

Symbol Tables: The Intuition

```
int x = 137;
int z = 42;
int MyFunction(int x, int y) {
  printf("%d,%d,%d\n", x@2, y@2, z@1);
  {
    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);
  }
}
```

Symbol Table	
x	**0**
z	**1**
x	**2**
y	**2**

Symbol Tables: The Intuition

```
int x = 137;
int z = 42;
int MyFunction(int x, int y) {
  printf("%d,%d,%d\n", x@2, y@2, z@1);
  {
    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);
  }
}
```

Symbol Table	
x	0
z	1
x	2
y	2

Symbol Tables: The Intuition

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

Symbol Table	
x	0
z	1
x	2
y	2
x	5
z	5

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@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

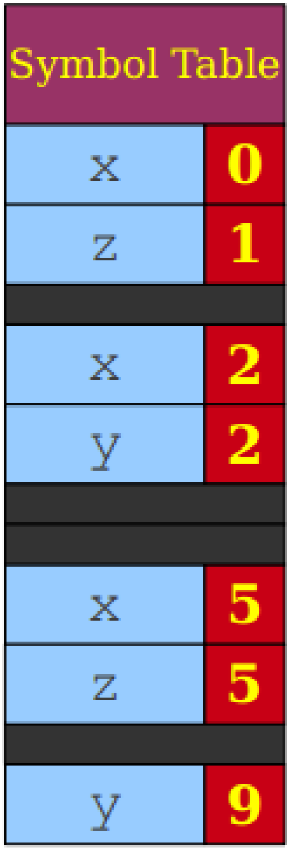
Symbol Tables: The Intuition

```
 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

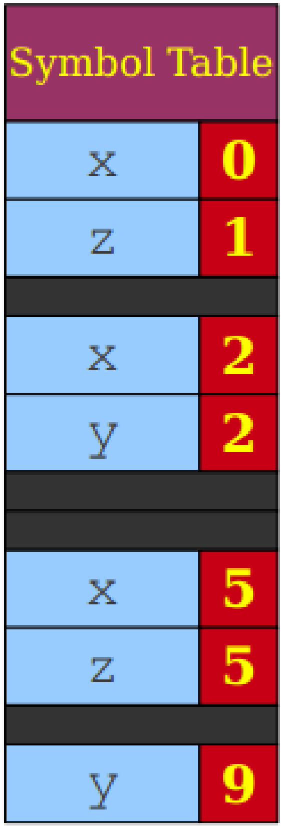
```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n",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

Symbol Tables: The Intuition

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



Symbol Table	
x	**0**
z	**1**
x	**2**
y	**2**
x	**5**
z	**5**
y	**9**

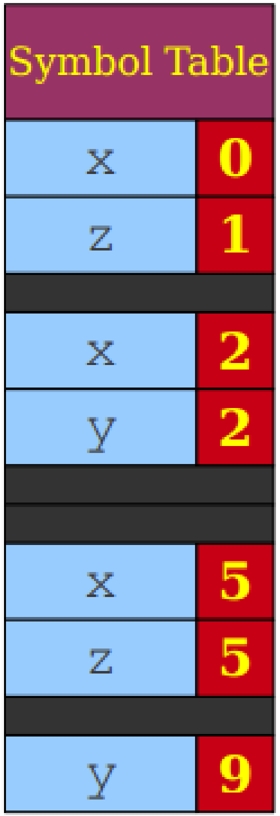
Symbol Tables: The Intuition

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

Symbol Table	
x	0
z	1
x	2
y	2
x	5
z	5
y	9

Symbol Tables: The Intuition

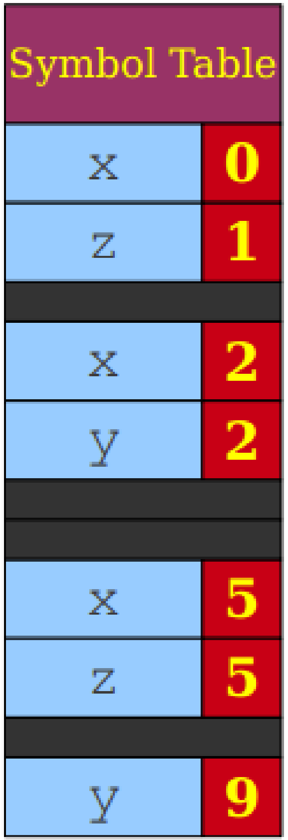
```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x@5, y@9, z@5);
12:       }
13:       printf("%d,%d,%d\n",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



Symbol Table	
x	0
z	1
x	2
y	2
x	5
z	5
y	9

Symbol Tables: The Intuition

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

Symbol Table Operations

- Typically implemented as a **stack of maps**.
- Each map corresponds to a particular scope.
- Stack allows for easy “enter” and “exit” operations.
- Symbol table operations are
  - **Push scope**: Enter a new scope.
  - **Pop scope**: Leave a scope, discarding all declarations in it.
  - **Insert symbol**: Add a new entry to the current scope.
  - **Lookup symbol**: Find what a name corresponds to.

Using a Symbol Table

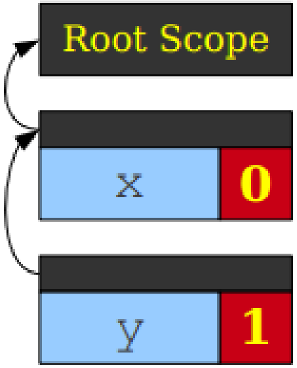
- To process a portion of the program that creates a scope (block statements, function calls, classes, etc.)
  - Enter a new scope.
  - Add all variable declarations to the symbol table.
  - Process the body of the block/function/class.
  - Exit the scope.
- Much of semantic analysis is defined in terms of recursive AST traversals like this.

Another View of Symbol Tables

```
int x;
int y;
int MyFunction(int x, int y)
{
  int w, z;
  {
    int y;
  }
  {
    int w;
  }
}
```

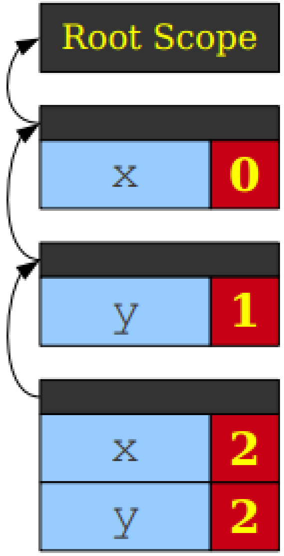
Another View of Symbol Tables

```
 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: }
```



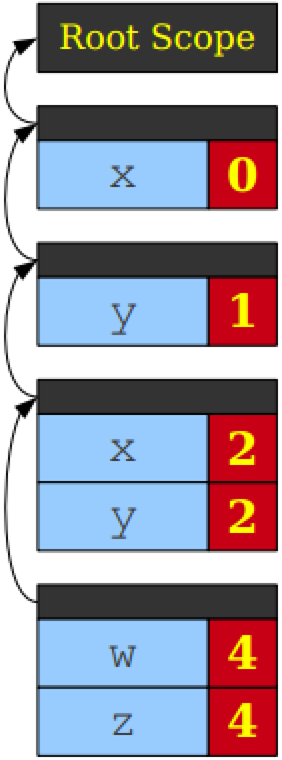
Another View of Symbol Tables

```
 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: }
```



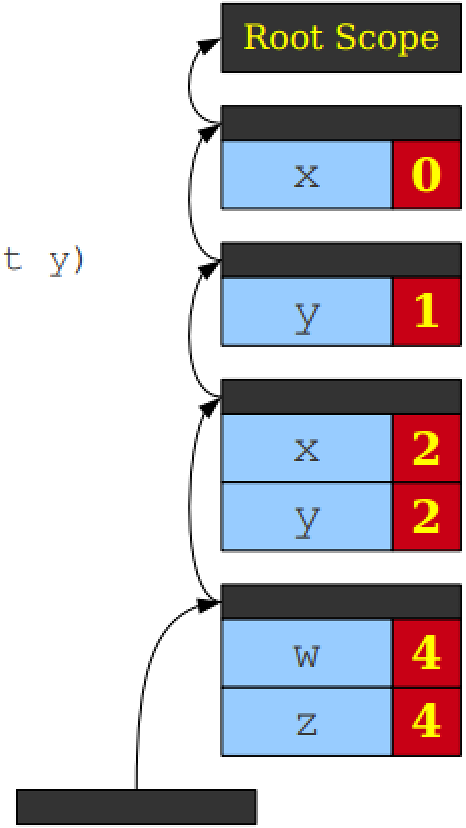
Another View of Symbol Tables

```
 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: }
```



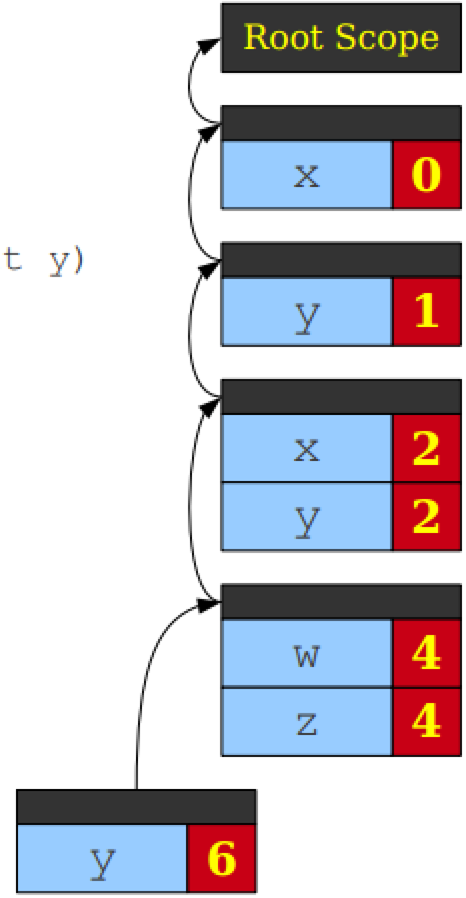
Another View of Symbol Tables

```
 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: }
```



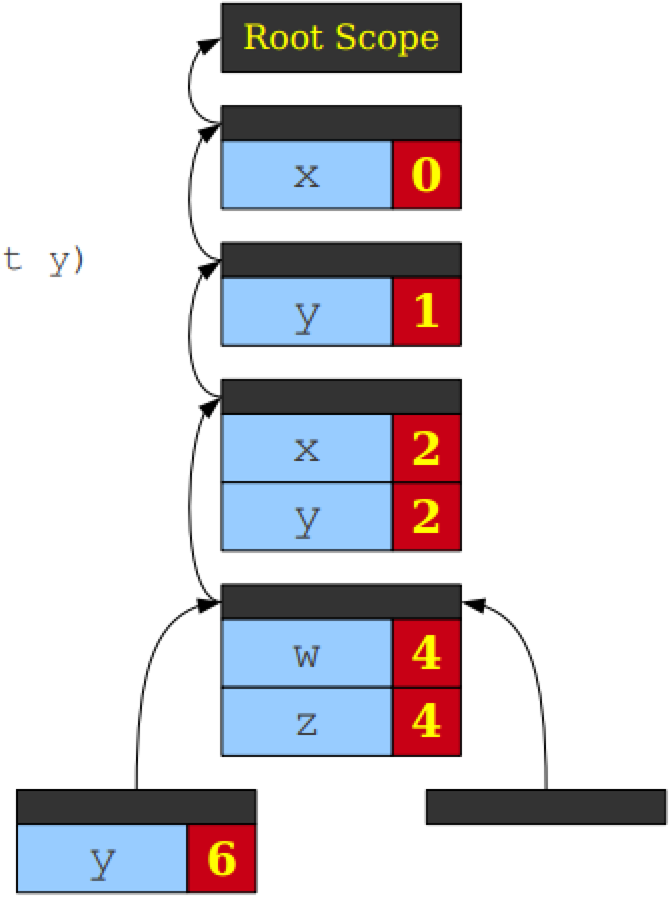
Another View of Symbol Tables

```
 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: }
```



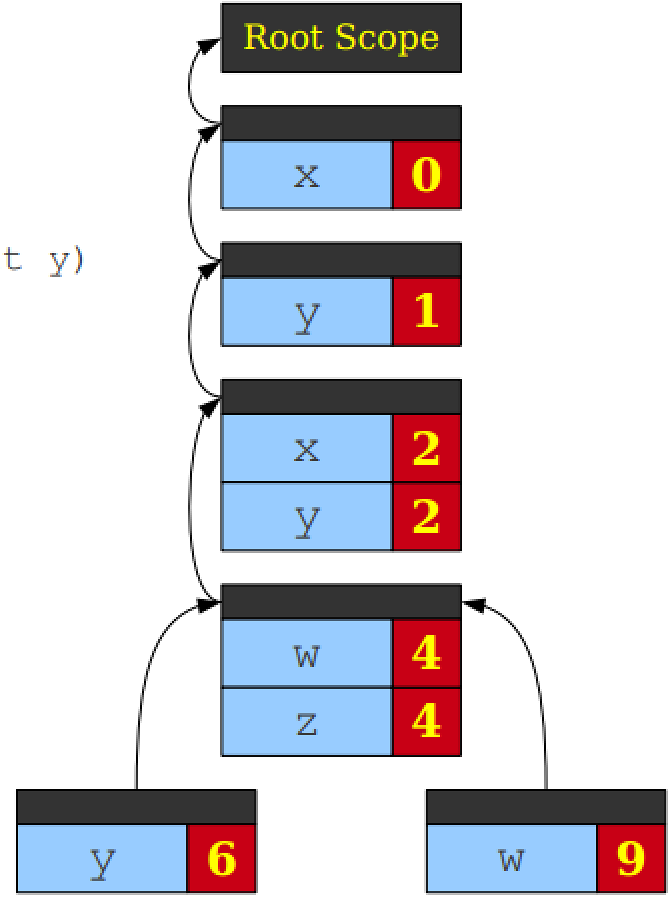
Another View of Symbol Tables

```
 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: }
```

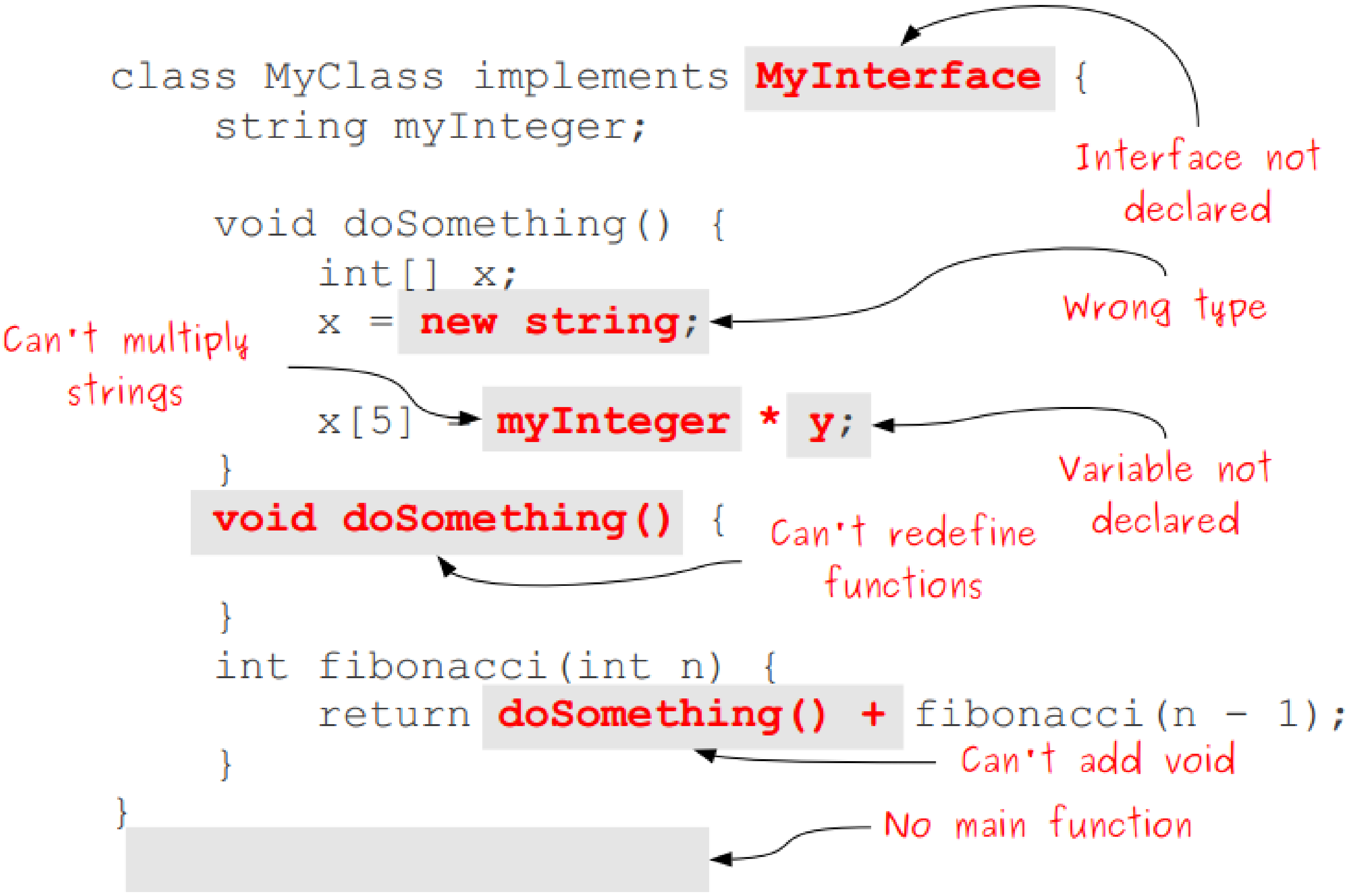


Spaghetti Stacks

- 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**.

Type-Checking

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



```
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

What is a Type?

- “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**.

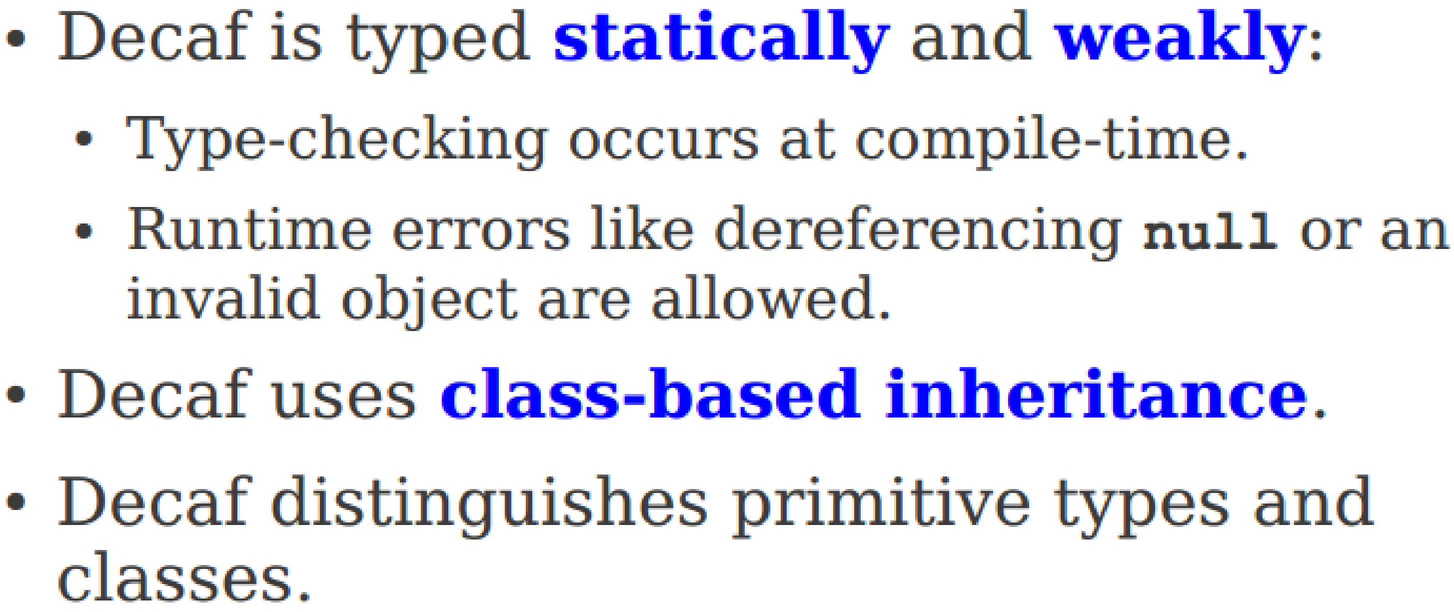
Types of Type-Checking

- **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.
  - (Why?)
- **No type checking.**
  - Throw caution to the wind!

Type Systems

- The rules governing permissible operations on types forms a **type system**.
- **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++

Typing in Decaf



- Decaf is typed **statically** and **weakly**:
  - Type-checking occurs at compile-time.
  - Runtime errors like dereferencing `null` or an invalid object are allowed.
- Decaf uses **class-based inheritance**.
- Decaf distinguishes primitive types and classes.

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.

An Example

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

Well-typed expression with wrong type.

An Example

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

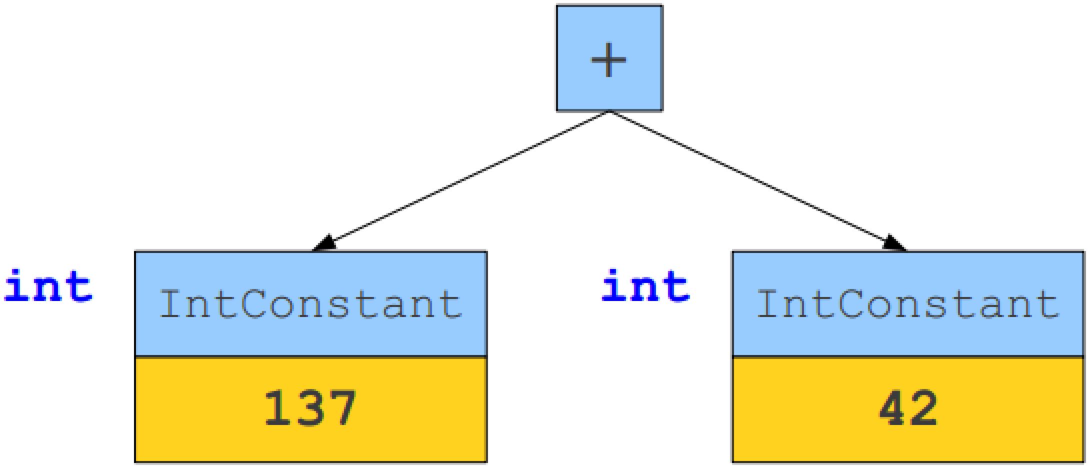
Expression with type error

Inferring Expression Types

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

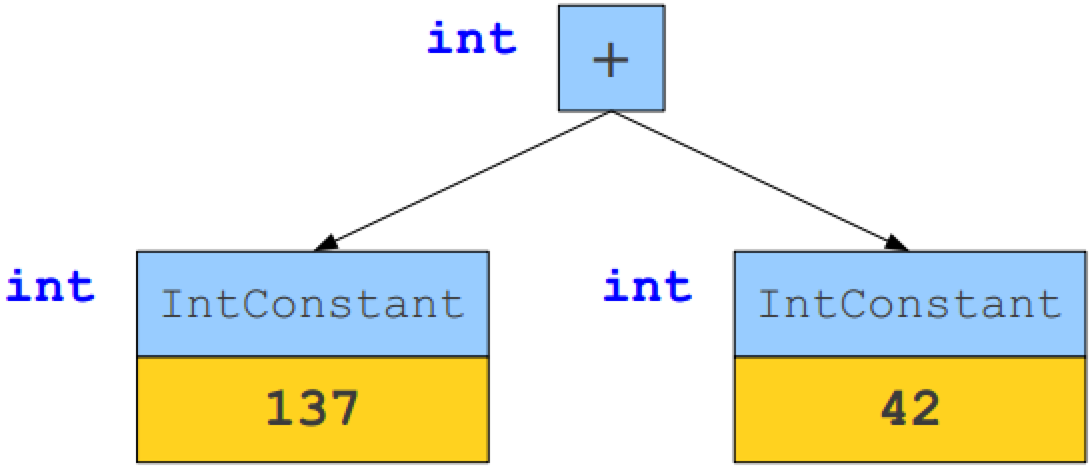
Inferring Expression Types

- 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**.



Type Checking as Proofs

- We can think of typing checking 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 a proof systems.

Sample Inference Rules

- “If **x** is an identifier that refers to an object of type **t**, the expression **x** has type **t**.”
- “If **e** is an integer constant, **e** has type **`int`**.”
- “If the operands $\mathbf{e}_1$ and $\mathbf{e}_2$ of $\mathbf{e}_1 + \mathbf{e}_2$ are known to have types **`int`** and **`int`**, then $\mathbf{e}_1 + \mathbf{e}_2$ has type **`int`**.”

Formalizing our Notation

- We will encode our axioms and inference rules using this syntax:

$$\frac{\text{Preconditions}}{\text{Postconditions}}$$

- This is read “if *preconditions* are true, we can infer *postconditions*.”

Examples of Formal Notation

$$\frac{\mathbf{A} \to \mathtt{t}\omega \text{ is a production.}}{\mathtt{t} \in \text{FIRST}(\mathbf{A})}$$

$$\frac{\mathbf{A} \to \varepsilon \text{ is a production.}}{\varepsilon \in \text{FIRST}(\mathbf{A})}$$

$$\frac{\begin{array}{c}\mathbf{A} \to \omega \text{ is a production.} \\ \mathtt{t} \in \text{FIRST}^*(\omega)\end{array}}{\mathtt{t} \in \text{FIRST}(\mathbf{A})}$$

$$\frac{\begin{array}{c}\mathbf{A} \to \omega \text{ is a production.} \\ \varepsilon \in \text{FIRST}^*(\omega)\end{array}}{\varepsilon \in \text{FIRST}(\mathbf{A})}$$

Formal Notation for Type Systems

- We write

$$\vdash \mathbf{e} : \mathbf{T}$$

  if the expression **e** has type **T**.
- The symbol $\vdash$ means "we can infer…"

Our Starting Axioms

$$\frac{}{\vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\vdash \texttt{false} : \texttt{bool}}$$

Some Simple Inference Rules

$$\frac{i \text{ is an integer constant}}{\vdash i : \texttt{int}}$$

$$\frac{s \text{ is a string constant}}{\vdash s : \texttt{string}}$$

$$\frac{d \text{ is a double constant}}{\vdash d : \texttt{double}}$$

More Complex Inference Rules

$$\frac{\begin{array}{c}\vdash e_1 : \texttt{int} \\ \vdash e_2 : \texttt{int}\end{array}}{\vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\begin{array}{c}\vdash e_1 : \texttt{double} \\ \vdash e_2 : \texttt{double}\end{array}}{\vdash e_1 + e_2 : \texttt{double}}$$

More Complex Inference Rules

If we can show that e_1 and e_2 have type int…

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

… then we can show that $e_1 + e_2$ has type int as well

Even More Complex Inference Rules

$$\frac{\begin{array}{c}\vdash e_1 : T \\ \vdash e_2 : T \\ T \text{ is a primitive type}\end{array}}{\vdash e_1 \;\texttt{==}\; e_2 : \texttt{bool}} \qquad \frac{\begin{array}{c}\vdash e_1 : T \\ \vdash e_2 : T \\ T \text{ is a primitive type}\end{array}}{\vdash e_1 \;\texttt{!=}\; e_2 : \texttt{bool}}$$

Strengthening our Inference Rules

- The facts we're proving have no *context*.
- We need to strengthen our inference rules to remember under what circumstances the results are valid.

Adding Scope

- We write

$$\mathbf{S} \vdash \mathbf{e} : \mathbf{T}$$

  if, in scope **S**, expression **e** has type **T**.
- Types are now proven relative to the scope they are in.

Old Rules Revisited

$$\frac{}{S \vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{S \vdash \texttt{false} : \texttt{bool}}$$

$$\frac{i \text{ is an integer constant}}{S \vdash i : \texttt{int}} \qquad \frac{s \text{ is a string constant}}{S \vdash s : \texttt{string}}$$

$$\frac{d \text{ is a double constant}}{S \vdash d : \texttt{double}}$$

$$\frac{\begin{array}{c} S \vdash e_1 : \texttt{double} \\ S \vdash e_2 : \texttt{double} \end{array}}{S \vdash e_1 + e_2 : \texttt{double}} \qquad \frac{\begin{array}{c} S \vdash e_1 : \texttt{int} \\ S \vdash e_2 : \texttt{int} \end{array}}{S \vdash e_1 + e_2 : \texttt{int}}$$

A Correct Rule

$$\frac{\begin{array}{c} x \text{ is an identifier.} \\ \boldsymbol{x} \textbf{ is a variable in scope S} \text{ with type T.} \end{array}}{S \vdash x : T}$$

Rules for Functions

f is an identifier.

$$\frac{}{S \vdash f(e_1, \ldots, e_n) \ : ??}$$

Rules for Functions

$$\dfrac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \end{array}}{S \vdash f(e_1, \ldots, e_n) \; : ??}$$

Rules for Functions

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \to U \end{array}}{S \vdash f(e_1, \ldots, e_n) \ : ??}$$

Rules for Functions

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : T_i \text{ for } 1 \le i \le n \end{array}}{S \vdash f(e_1, \ldots, e_n) : ??}$$

Rules for Functions

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : T_i \text{ for } 1 \leq i \leq n \end{array}}{S \vdash f(e_1, \ldots, e_n) \ : U}$$

Rules for Arrays

$$\frac{S \vdash e_1 : T\texttt{[]} \qquad S \vdash e_2 : \texttt{int}}{S \vdash e_1[e_2] : T}$$

Rule for Assignment

$$\frac{\begin{array}{c} S \vdash e_1 : T \\ S \vdash e_2 : T \end{array}}{S \vdash e_1 = e_2 : T}$$

Typing with Classes

- How do we factor inheritance into our inference rules?
- We need to consider the shape of class hierarchies.

Rule for Assignment

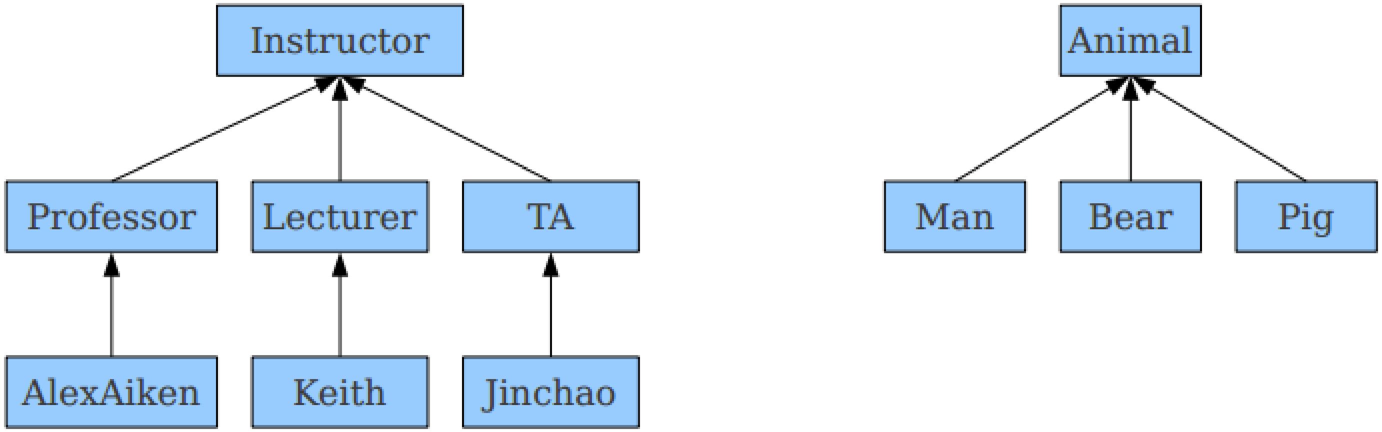
$$\frac{S \vdash e_1 : T \qquad S \vdash e_2 : T}{S \vdash e_1 = e_2 : T}$$

If **`Derived`** extends **`Base`**, will this rule work for this code?

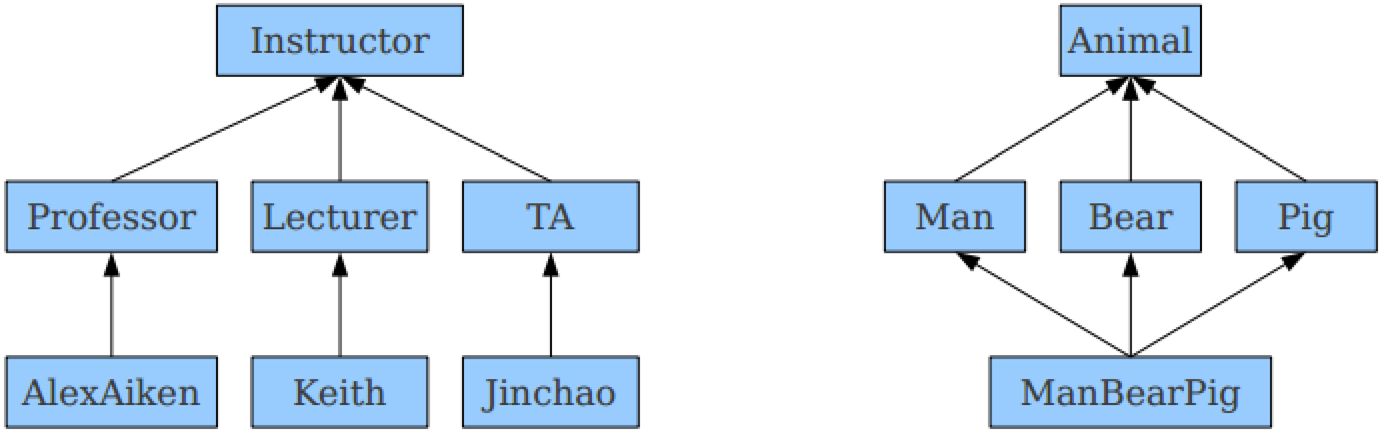
```
Base    myBase;
Derived myDerived;

myBase = myDerived;
```

Single Inheritance



Multiple Inheritance



Instructor

Professor

Lecturer

TA

AlexAiken

Keith

Jinchao

Animal

Man

Bear

Pig

ManBearPig

Properties of Inheritance Structures

- Any type is convertible to itself. (**reflexivity**)
- If A is convertible to B and B is convertible to C, then A is convertible to C. (**transitivity**)
- If A is convertible to B and B is convertible to A, then A and B are the same type. (**antisymmetry**)
- This defines a **partial order** over types.

Types and Partial Orders

- We say that $A \leq B$ if A is convertible to B.
- We have that
  - $A \leq A$
  - $A \leq B$ and $B \leq C$ implies $A \leq C$
  - $A \leq B$ and $B \leq A$ implies $A = B$

Updated Rule for Assignment

$$\frac{\begin{array}{c} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_2 \leq T_1 \end{array}}{S \vdash e_1 = e_2 : T_1}$$

Can we do better than this?

Updated Rule for Assignment

$$\frac{\begin{array}{c} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_2 \leq T_1 \end{array}}{S \vdash e_1 = e_2 : \mathbf{T_2}}$$

Updated Rule for Assignment

$$\frac{\begin{array}{c} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_2 \leq T_1 \end{array}}{S \vdash e_1 = e_2 : \mathbf{T_2}}$$

Updated Rule for Comparisons

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

Updated Rule for Comparisons

$$\frac{\begin{array}{c} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 == e_2 : \texttt{bool}}$$

$$\frac{\begin{array}{c} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \text{ and } T_2 \text{ are of class type.} \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash e_1 == e_2 : \texttt{bool}}$$

Updated Rule for Comparisons

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

$$\frac{S \vdash e_1 : T_1 \qquad S \vdash e_2 : T_2 \qquad T_1 \text{ and } T_2 \text{ are of class type.} \qquad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash e_1 == e_2 : \texttt{bool}}$$

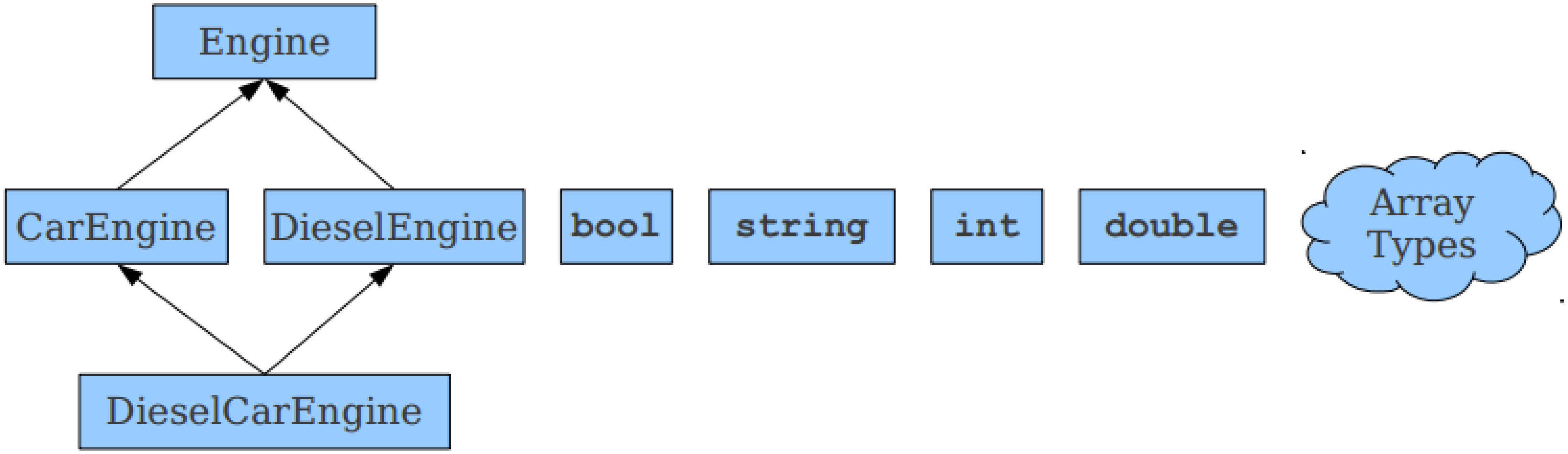
Updated Rule for Comparisons

Can we unify these rules?

$$\frac{\begin{array}{c} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 == e_2 : \texttt{bool}}$$

$$\frac{\begin{array}{c} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \text{ and } T_2 \text{ are of class type.} \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash e_1 == e_2 : \texttt{bool}}$$

The Shape of Types



Engine

CarEngine

DieselEngine

DieselCarEngine

`bool`

`string`

`int`

`double`

Array Types

Extending Convertibility

- If A is a primitive or array type, A is only convertible to itself.
- More formally, if A and B are types and A is a primitive or array type:
  - $A \leq B$ implies $A = B$
  - $B \leq A$ implies $A = B$

Updated Rule for Comparisons

$$\frac{\begin{array}{c} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 == e_2 : \texttt{bool}} \qquad \frac{\begin{array}{c} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \text{ and } T_2 \text{ are of class type.} \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash e_1 == e_2 : \texttt{bool}}$$

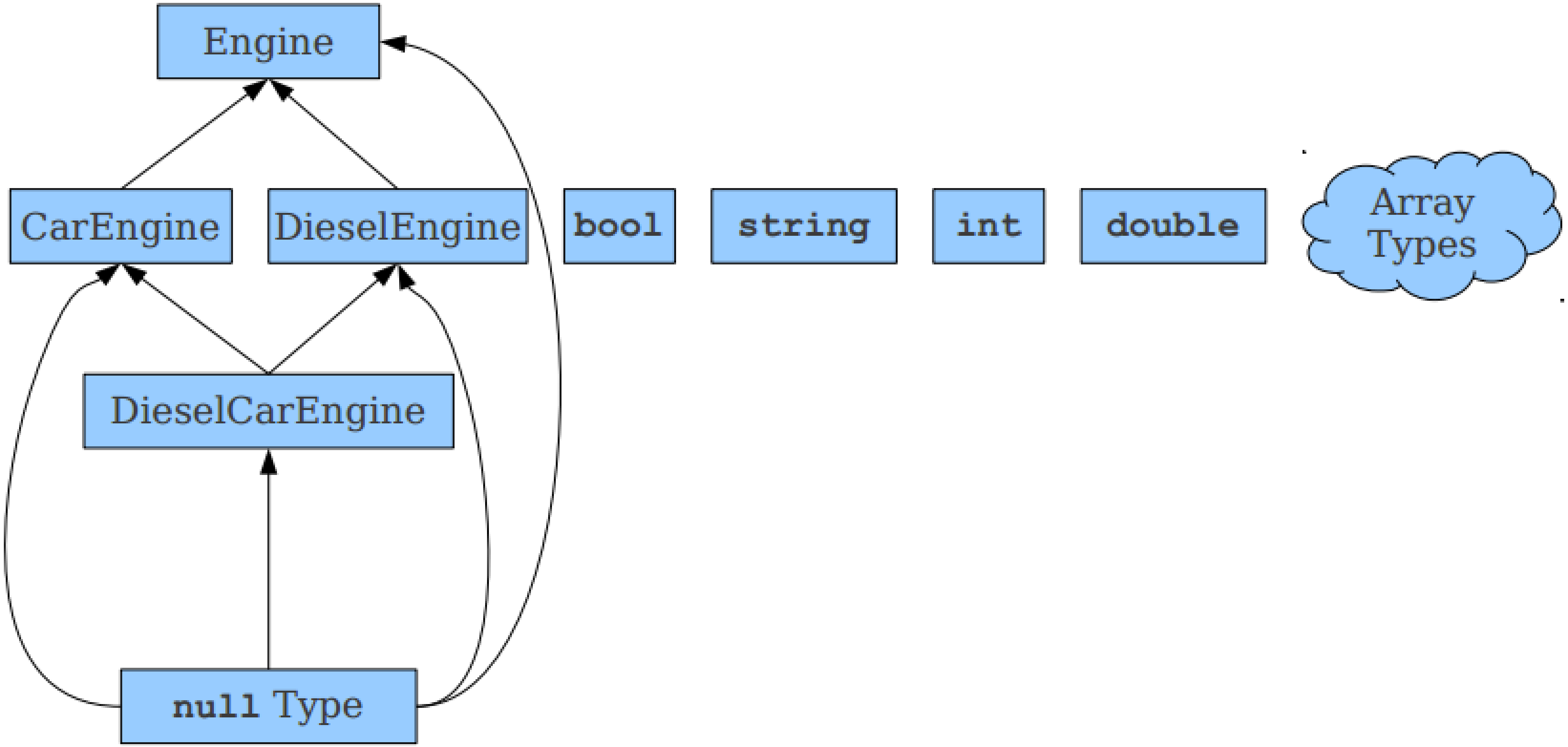
$$\begin{array}{c} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}$$

Updated Rule for Function Calls

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope } S. \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : R_i \ \text{ for } 1 \leq i \leq n \\ R_i \leq T_i \ \text{ for } 1 \leq i \leq n \end{array}}{S \vdash f(e_1, \ldots, e_n) \ : U}$$

A Tricky Case

$$\frac{\qquad\qquad\qquad\qquad}{S \vdash \texttt{\textbf{null}} : ??}$$


Engine
CarEngine
DieselEngine
bool
string
int
double
Array Types
DieselCarEngine
null Type

Handling `null`

- Define a new type corresponding to the type of the literal `null`; call it "**`null` type**."
- Define `null` type ≤ A for any class type A.
- The `null` type is (typically) not accessible to programmers; it's only used internally.
- Many programming languages have types like these.

A Tricky Case

$$\frac{}{S \vdash \texttt{null} : \texttt{null}\ \text{type}}$$

Object-Oriented Considerations

$$\frac{S \text{ is in scope of class } T.}{S \vdash \mathbf{\texttt{this}} : T}$$

$$\frac{T \text{ is a class type.}}{S \vdash \mathbf{\texttt{new}}\ T : T} \qquad \frac{S \vdash e : \mathbf{\texttt{int}}}{S \vdash \mathbf{\texttt{NewArray}}\texttt{(}e, T\texttt{)} : T\mathbf{\texttt{[]}}}$$

Using our Type Proofs

- We can now prove the types of various expressions.
- How do we check…
  - … that **`if`** statements have well-formed conditional expressions?
  - … that **`return`** statements actually return the right type of value?
- Use another proof system!

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }
```

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

Expression with type error

Proofs of Structural Soundness

- Idea: extend our proof system to statements to confirm that they are well-formed.
- We say that

$$S \vdash \mathrm{WF}(\mathit{stmt})$$

  if the statement *stmt* is **well-formed** in scope S.
- The type system is satisfied if for every function *f* with body B in scope S, we can show $S \vdash \mathrm{WF}(B)$.

A Simple Well-Formedness Rule

$$\frac{S \vdash \textit{expr} : \mathrm{T}}{S \vdash \mathrm{WF}(\textit{expr};)}$$

If we can assign a valid type to an expression in scope S…

… then it is a valid statement in scope S.

A More Complex Rule

$$\frac{\begin{array}{c} \mathrm{S} \vdash \mathrm{WF}(\mathit{stmt}_1) \\ \mathrm{S} \vdash \mathrm{WF}(\mathit{stmt}_2) \end{array}}{\mathrm{S} \vdash \mathrm{WF}(\mathit{stmt}_1\ \mathit{stmt}_2)}$$

A Rule for Loops

$$\frac{\begin{array}{c} S \vdash \mathit{expr} : \texttt{bool} \\ \text{S' is the scope inside the loop.} \\ S' \vdash \mathrm{WF}(\mathit{stmt}) \end{array}}{S \vdash \mathrm{WF}(\texttt{while (}\mathit{expr}\texttt{)}\ \mathit{stmt})}$$

Rules for `return`

$$\frac{\begin{array}{c}\text{S is in a function returning T}\\ S \vdash \mathit{expr} : T'\\ T' \leq T\end{array}}{S \vdash \mathrm{WF}(\texttt{return}\ \mathit{expr};)}$$

$$\frac{\text{S is in a function returning } \texttt{void}}{S \vdash \mathrm{WF}(\texttt{return};)}$$

Checking Well-Formedness

- Recursively walk the AST.
- For each statement:
  - Typecheck any subexpressions it contains.
    - Report errors if **no** type can be assigned.
    - Report errors if the **wrong** type is assigned.
  - Typecheck child statements.
  - Check the overall correctness.