

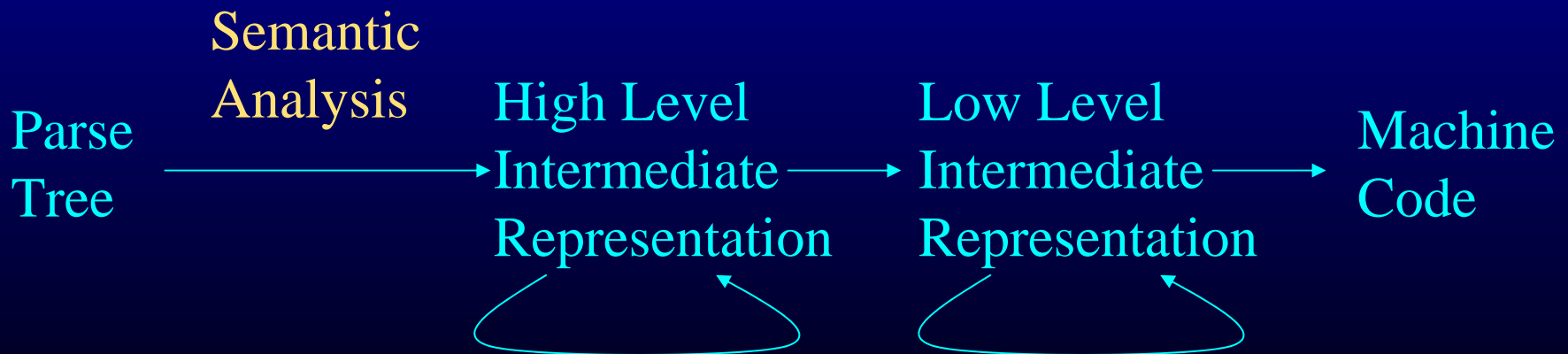
MIT 6.035  
Intermediate Formats

Martin Rinard

Laboratory for Computer Science  
Massachusetts Institute of Technology

# Program Representation Goals

- Enable Program Analysis and Transformation
  - Semantic Checks, Correctness Checks, Optimizations
- Structure Translation to Machine Code
  - Sequence of Steps



# High Level IR

- Preserves Object Structure
- Preserves Structured Flow of Control
- Primary Goal: Analyze Program

# Low Level IR

- Moves Data Model to Flat Address Space
- Eliminates Structured Control Flow
- Suitable for Low Level Compilation Tasks
  - Register Allocation
  - Instruction Selection

# Examples of Object Representation and Program Execution

(This happens when program runs)

# Example Vector Class

```
class vector {  
    int v[];  
    void add(int x) {  
        int i;  
        i = 0;  
        while (i < v.length) { v[i] = v[i]+x; i = i+1; }  
    }  
}
```

# Representing Arrays

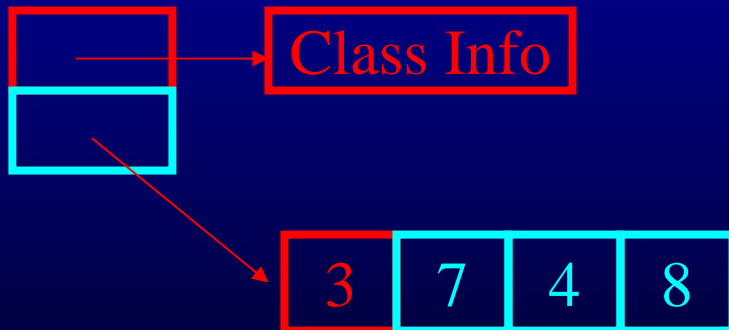
- Items Stored Contiguously In Memory
- Length Stored In First Word



- Color Code
  - Red - generated by compiler automatically
  - Blue, Yellow, Lavender - program data or code
  - Magenta - executing code or data

# Representing Vector Objects

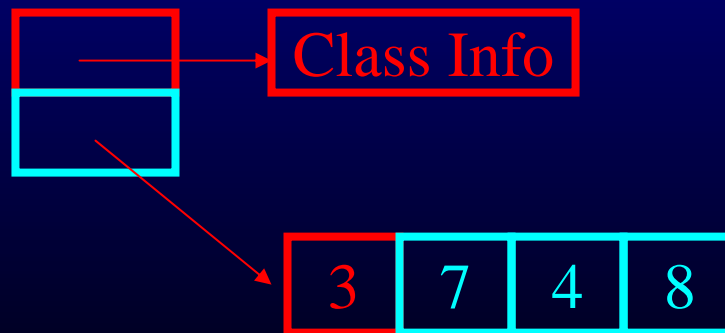
- First Word Points to Class Information
  - Method Table, Garbage Collector Data
- Next Words Have Object Fields
  - For vectors, Next Word is Reference to Array



# Invoking Vector Add Method

```
vect.add(1);
```

- Create Activation Record

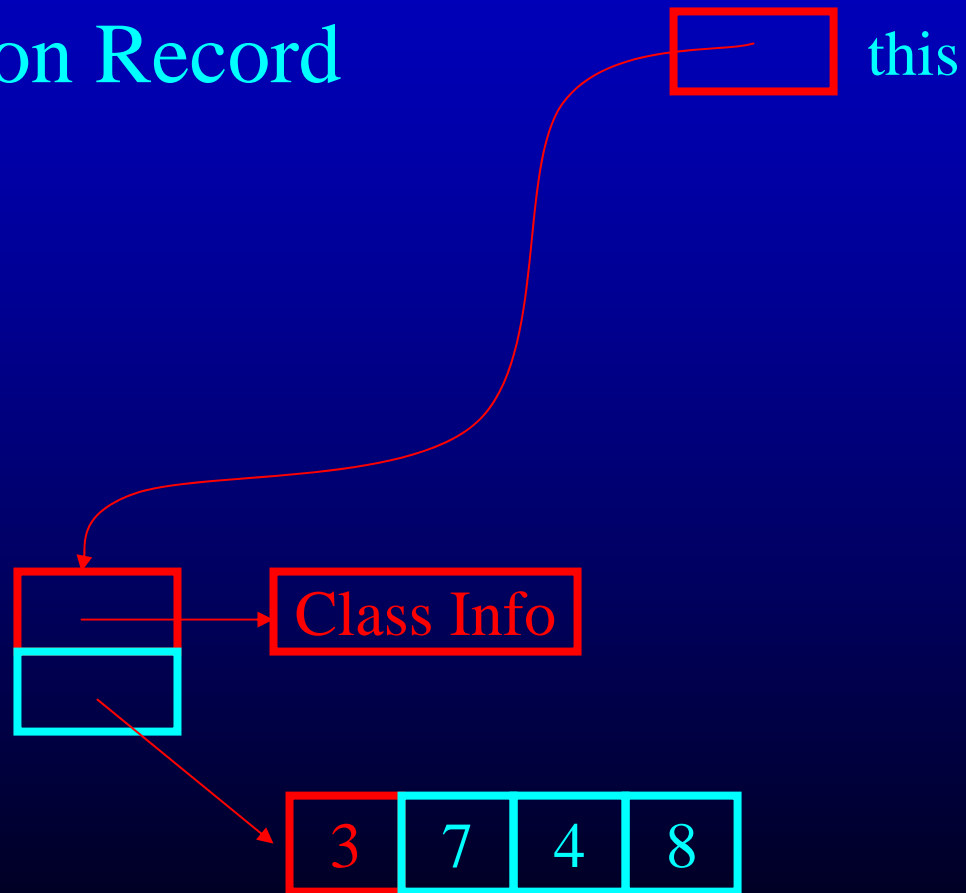




# Invoking Vector Add Method

vect.add(1);

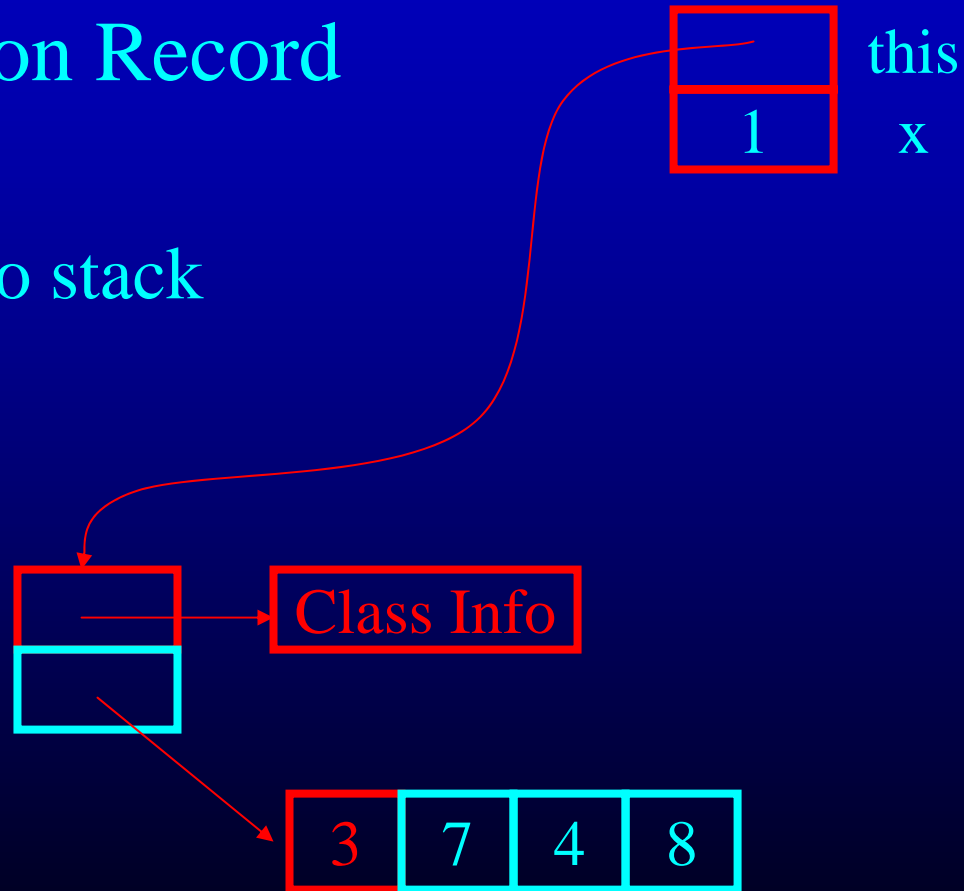
- Create Activation Record
  - this onto stack



# Invoking Vector Add Method

vect.add(1);

- Create Activation Record
  - this onto stack
  - parameters onto stack

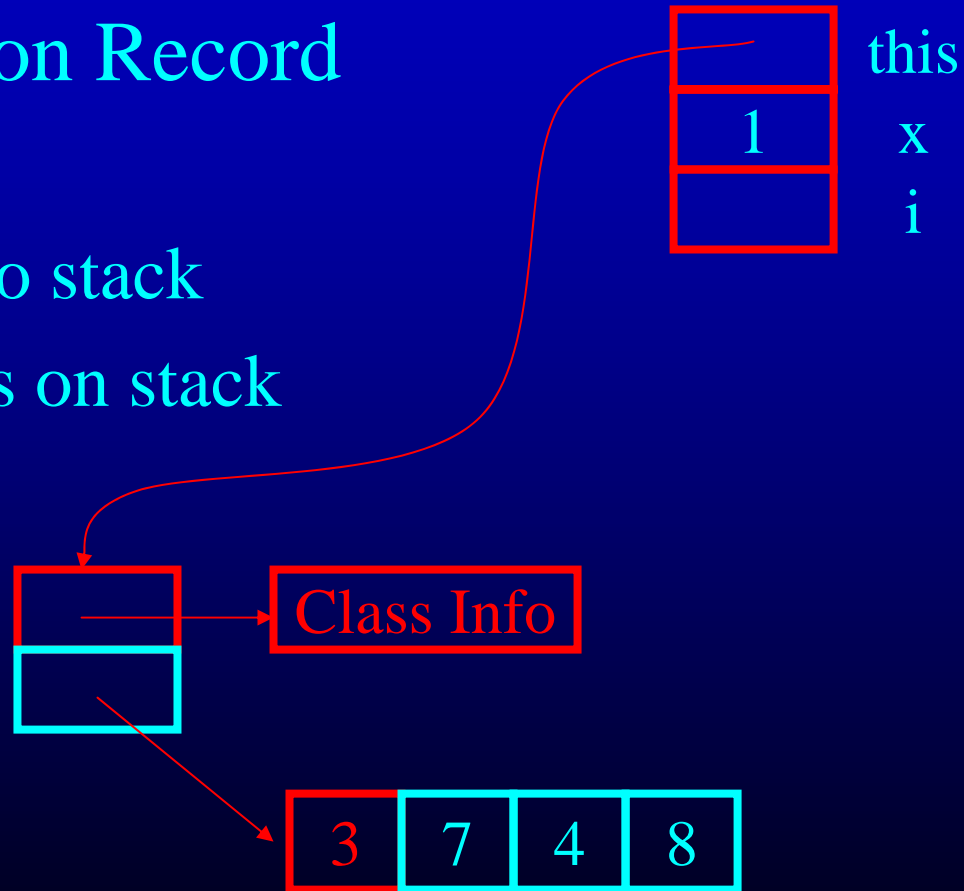


# Invoking Vector Add Method

vect.add(1);

- Create Activation Record

- this onto stack
- parameters onto stack
- space for locals on stack



# Executing Vector Add Method

```
void add(int x) {
```

```
    int i;
```

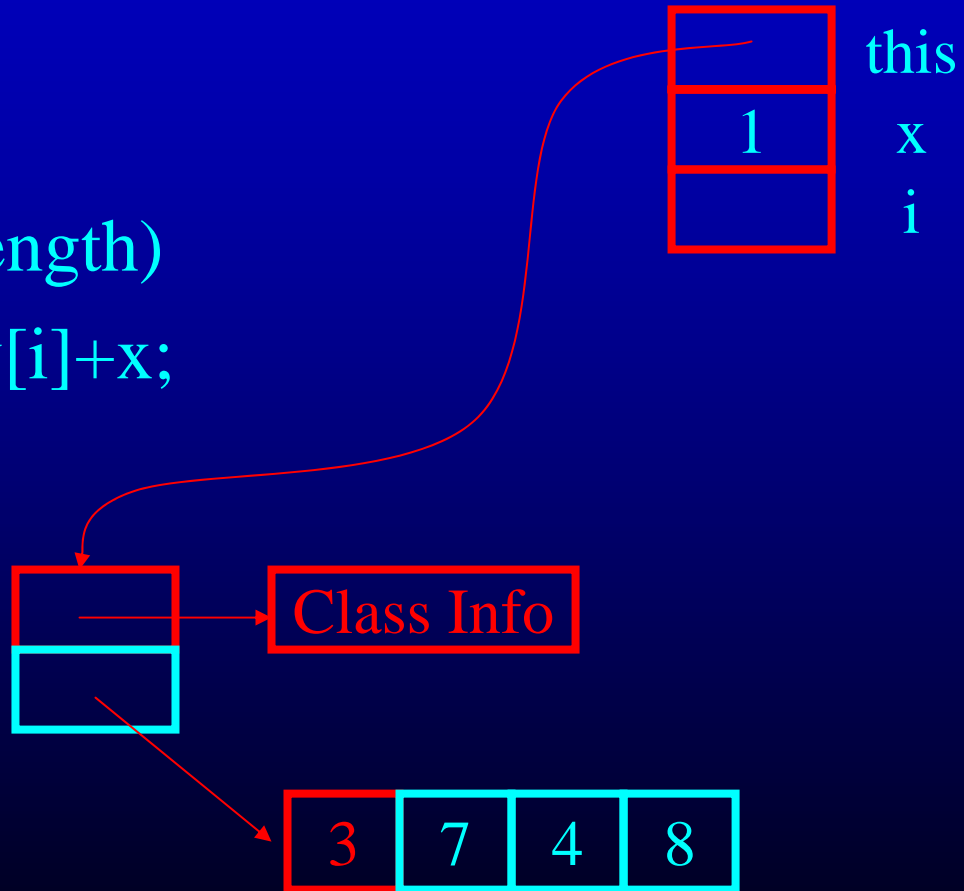
```
    i = 0;
```

```
    while (i < v.length)
```

```
        v[i] = v[i]+x;
```

```
        i = i+1;
```

```
}
```



# Executing Vector Add Method

```
void add(int x) {
```

```
    int i;
```

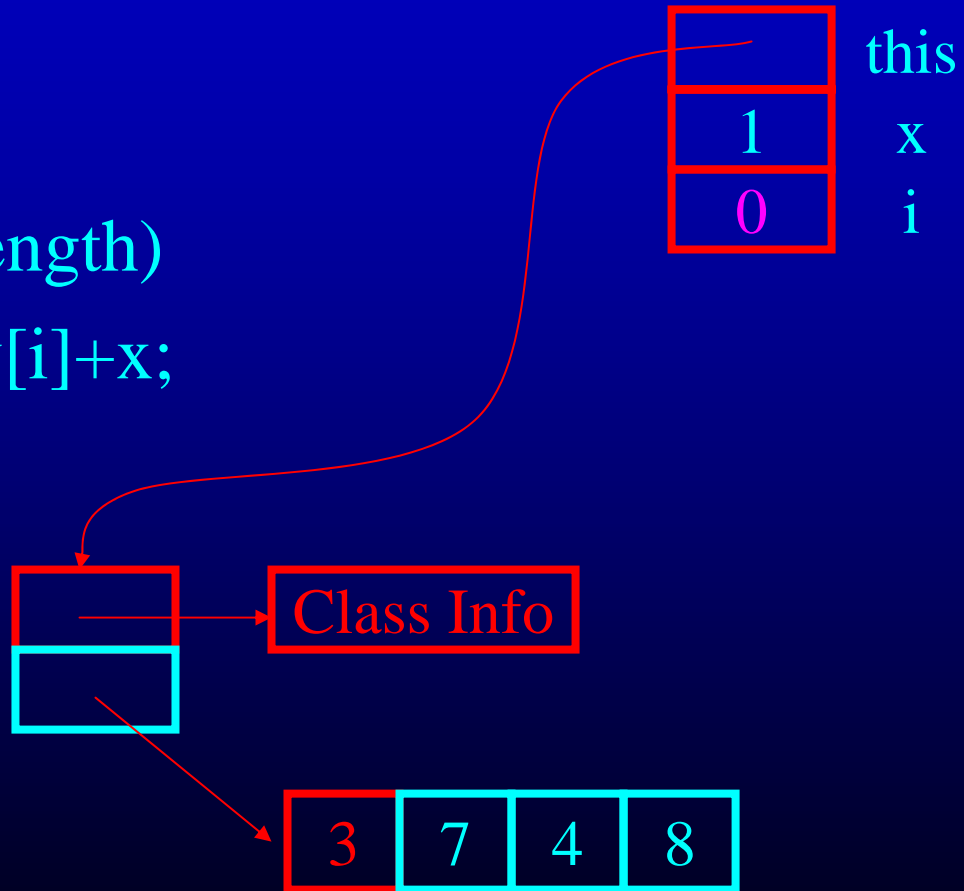
```
    i = 0;
```

```
    while (i < v.length)
```

```
        v[i] = v[i]+x;
```

```
        i = i+1;
```

```
}
```



# Executing Vector Add Method

```
void add(int x) {
```

```
    int i;
```

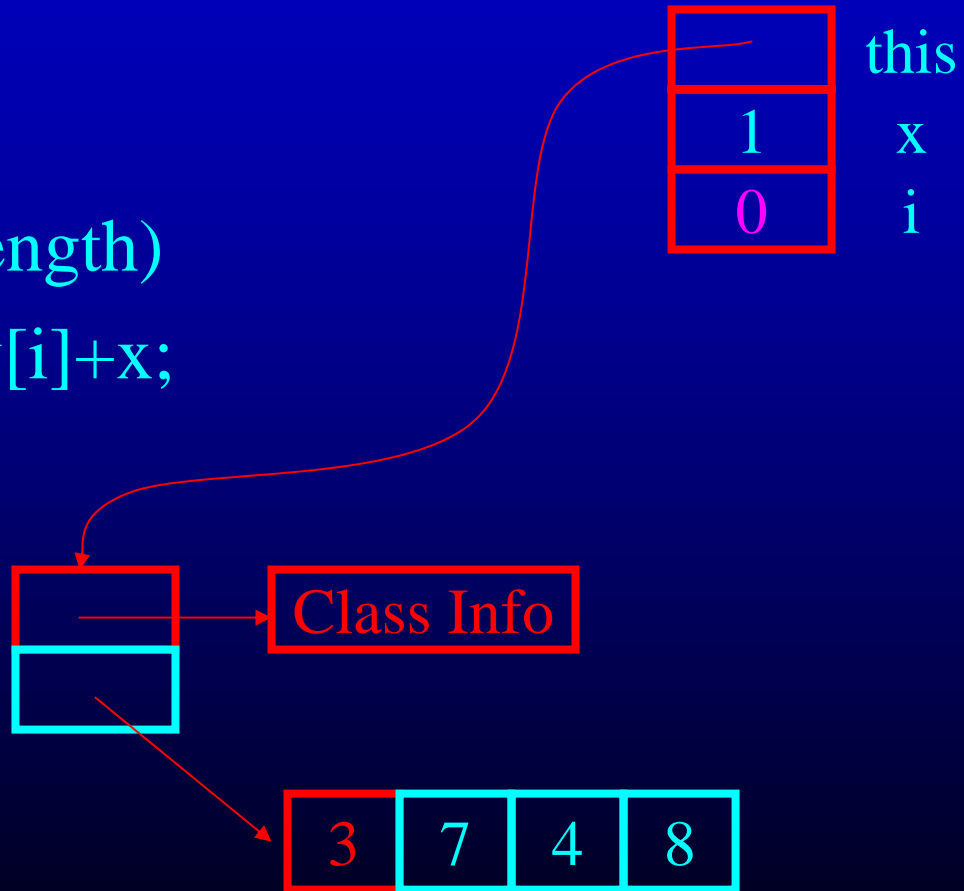
```
    i = 0;
```

```
    while (i < v.length)
```

```
        v[i] = v[i]+x;
```

```
        i = i+1;
```

```
}
```



# Executing Vector Add Method

```
void add(int x) {
```

```
    int i;
```

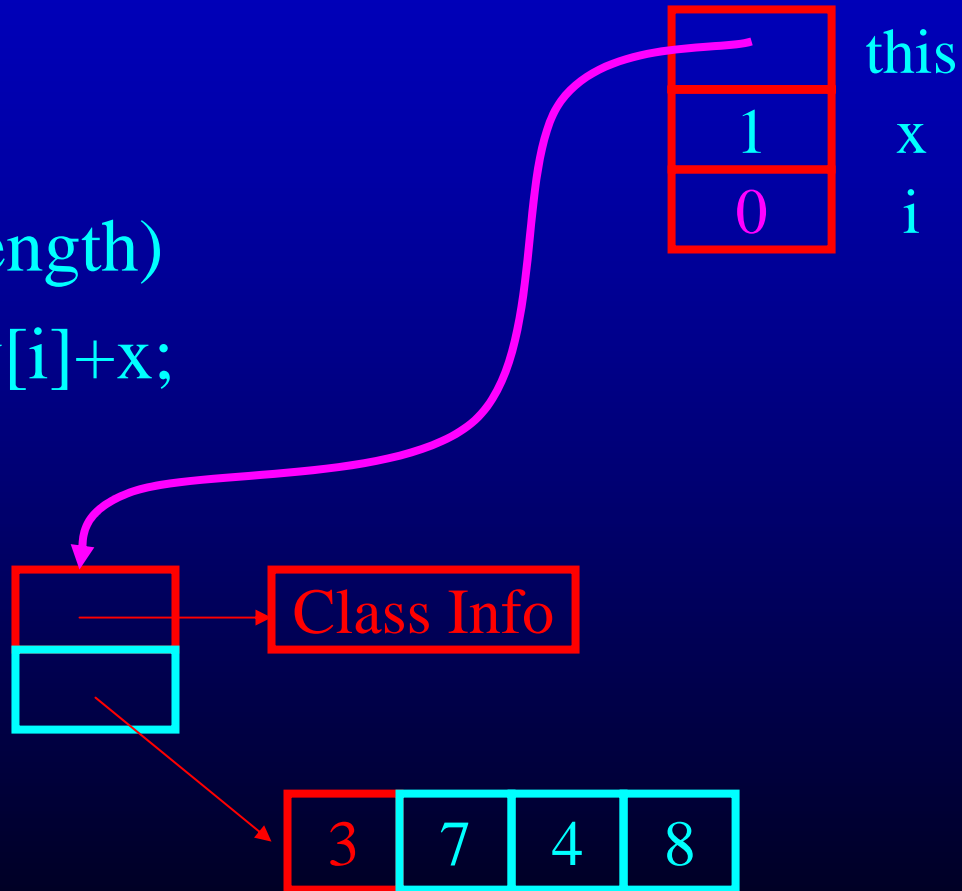
```
    i = 0;
```

```
    while (i < v.length)
```

```
        v[i] = v[i]+x;
```

```
        i = i+1;
```

```
}
```



# Executing Vector Add Method

```
void add(int x) {
```

```
    int i;
```

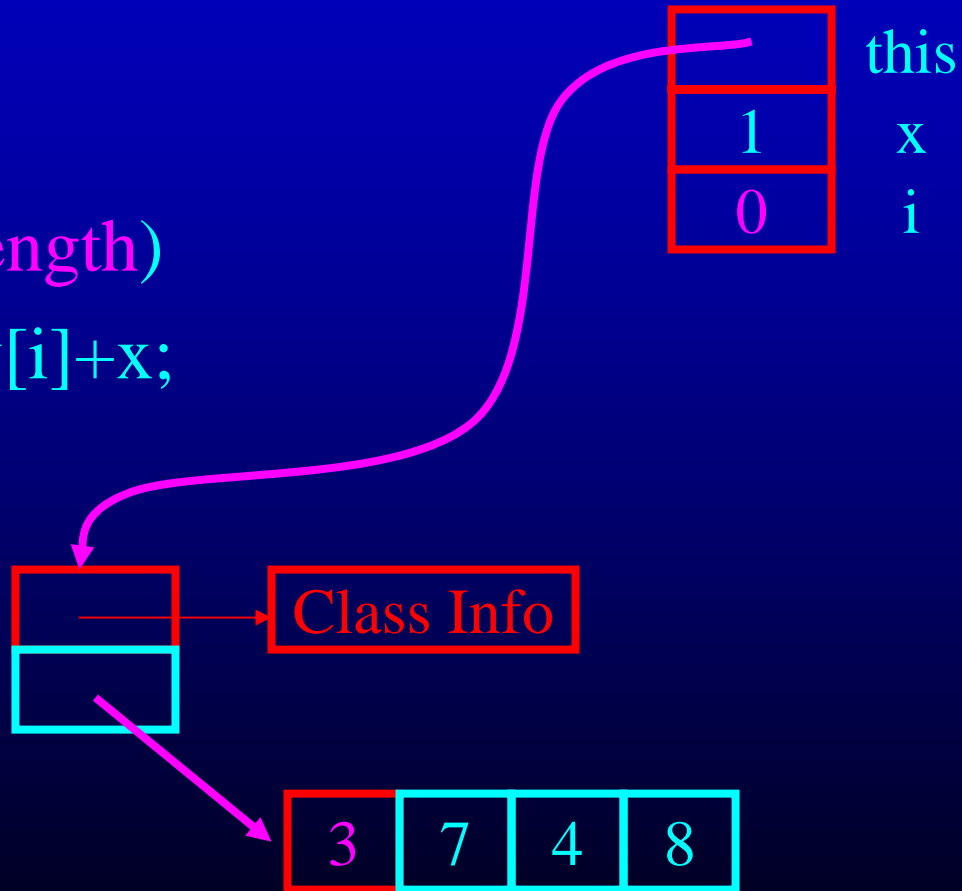
```
    i = 0;
```

```
    while (i < v.length)
```

```
        v[i] = v[i]+x;
```

```
        i = i+1;
```

```
}
```





# Executing Vector Add Method

```
void add(int x) {
```

```
    int i;
```

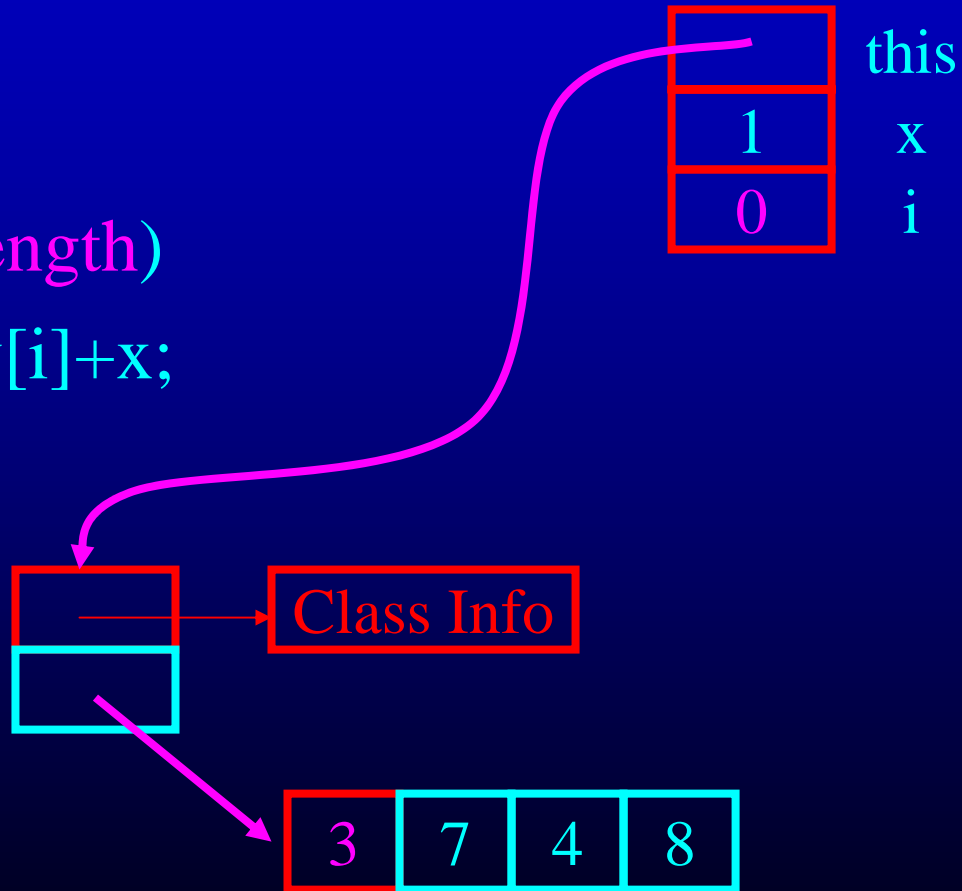
```
    i = 0;
```

```
    while (i < v.length)
```

```
        v[i] = v[i]+x;
```

```
        i = i+1;
```

```
}
```



# Executing Vector Add Method

```
void add(int x) {
```

```
    int i;
```

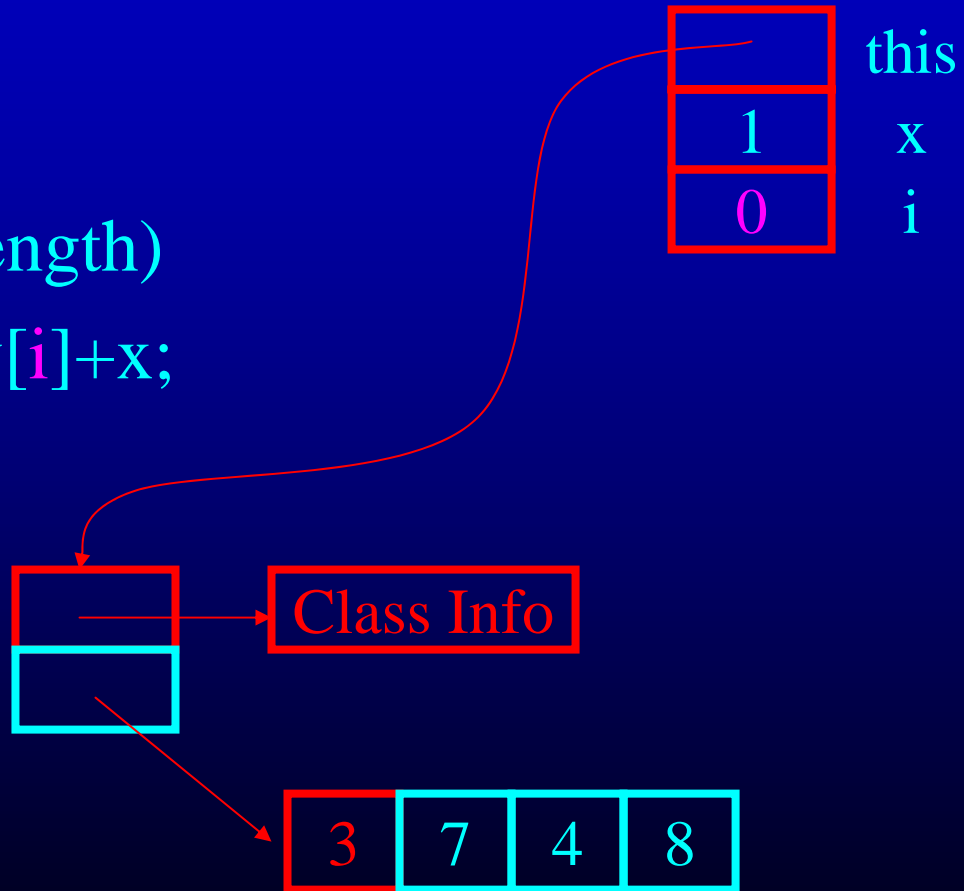
```
    i = 0;
```

```
    while (i < v.length)
```

```
        v[i] = v[i]+x;
```

```
        i = i+1;
```

```
}
```



# Executing Vector Add Method

```
void add(int x) {
```

```
    int i;
```

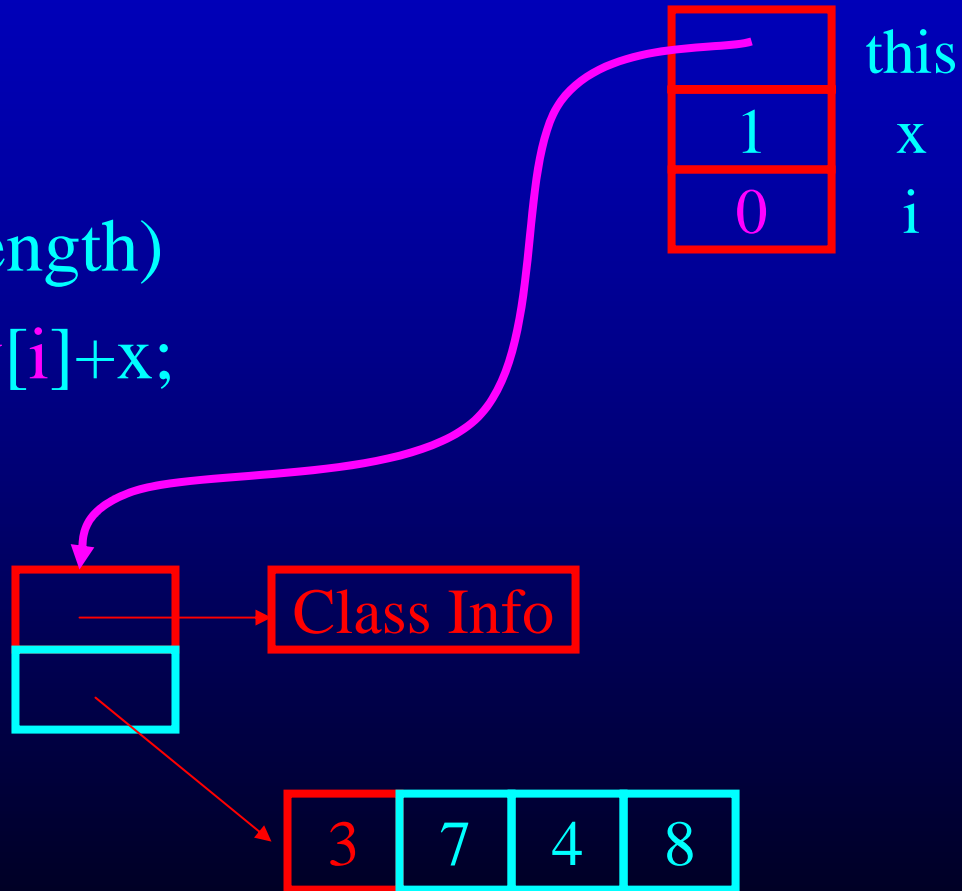
```
    i = 0;
```

```
    while (i < v.length)
```

```
        v[i] = v[i]+x;
```

```
        i = i+1;
```

```
}
```



# Executing Vector Add Method

```
void add(int x) {
```

```
    int i;
```

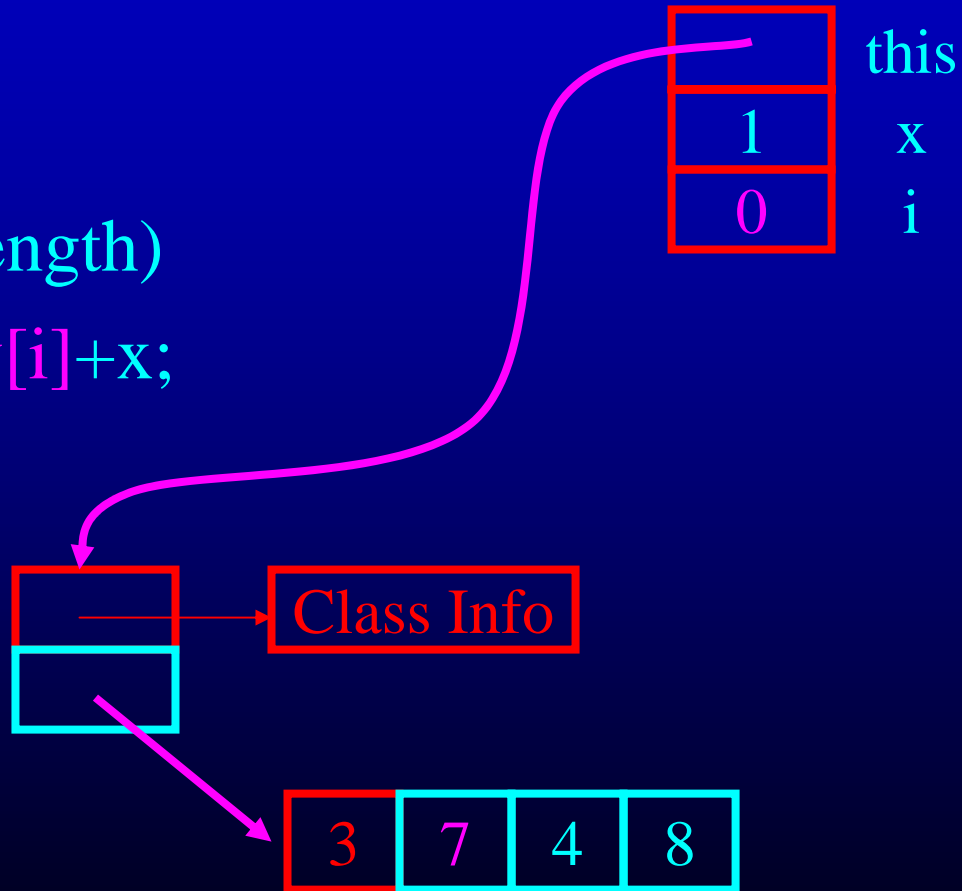
```
    i = 0;
```

```
    while (i < v.length)
```

```
        v[i] = v[i]+x;
```

```
        i = i+1;
```

```
}
```



# Executing Vector Add Method

```
void add(int x) {
```

```
    int i;
```

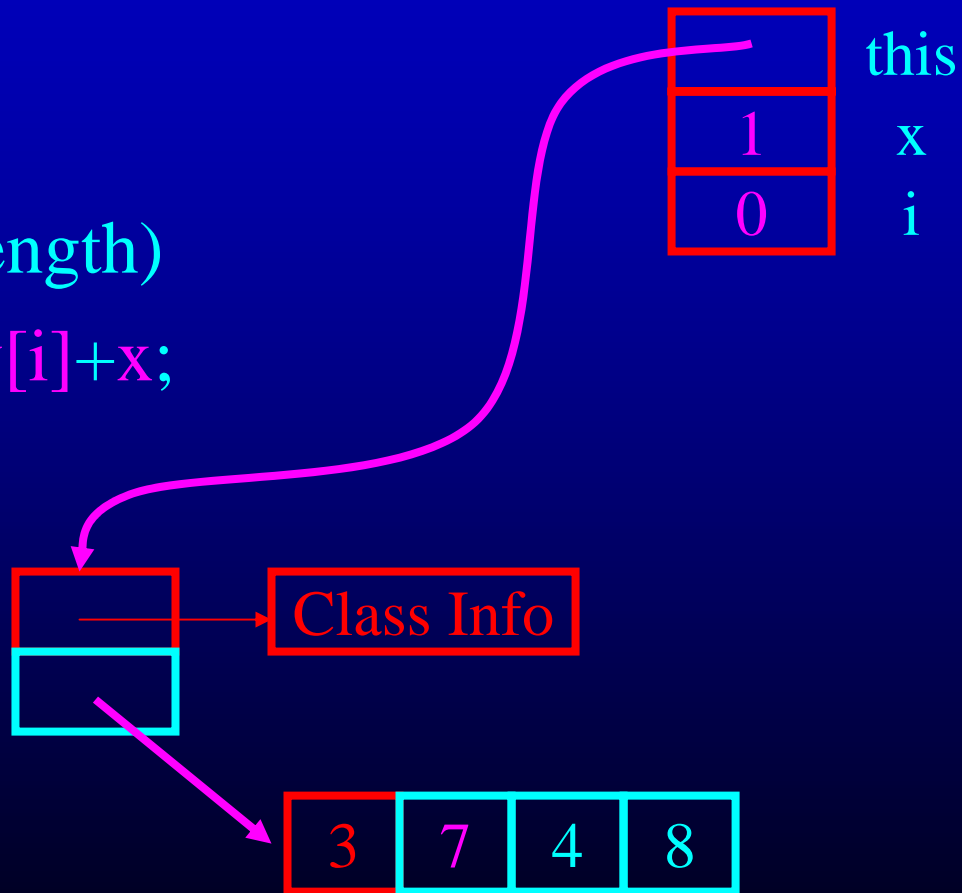
```
    i = 0;
```

```
    while (i < v.length)
```

```
        v[i] = v[i]+x;
```

```
        i = i+1;
```

```
}
```



# Executing Vector Add Method

```
void add(int x) {
```

```
    int i;
```

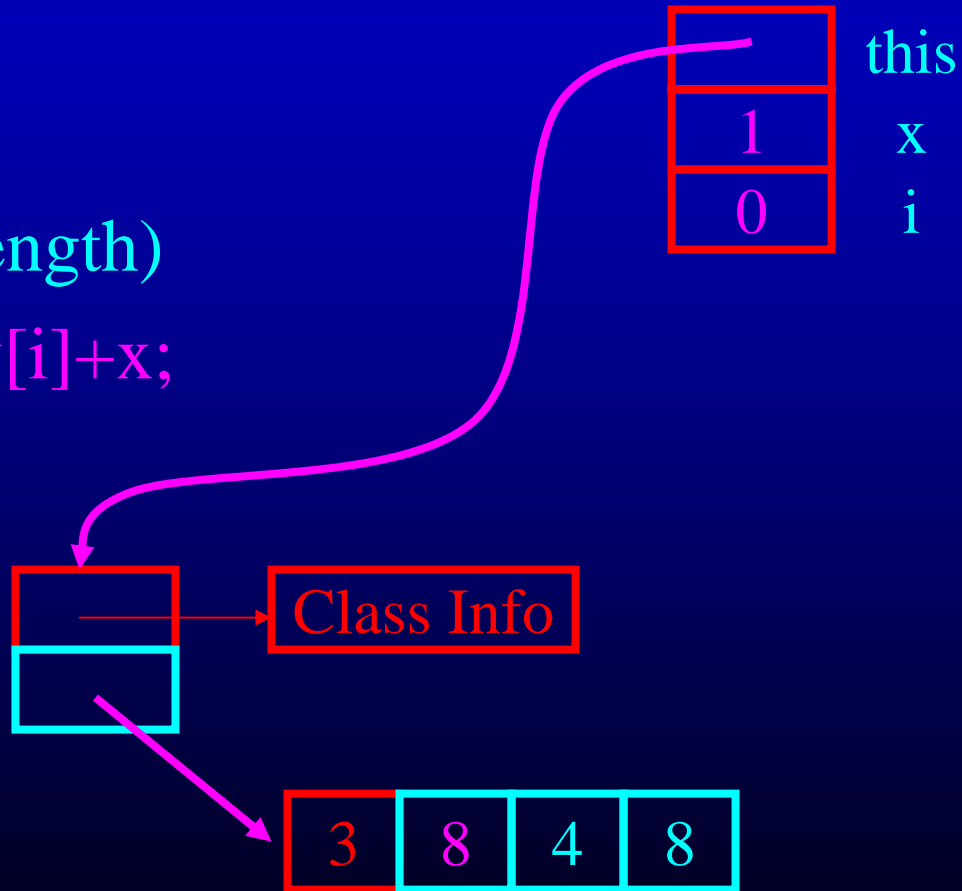
```
    i = 0;
```

```
    while (i < v.length)
```

```
        v[i] = v[i]+x;
```

```
        i = i+1;
```

```
}
```



# Executing Vector Add Method

```
void add(int x) {
```

```
    int i;
```

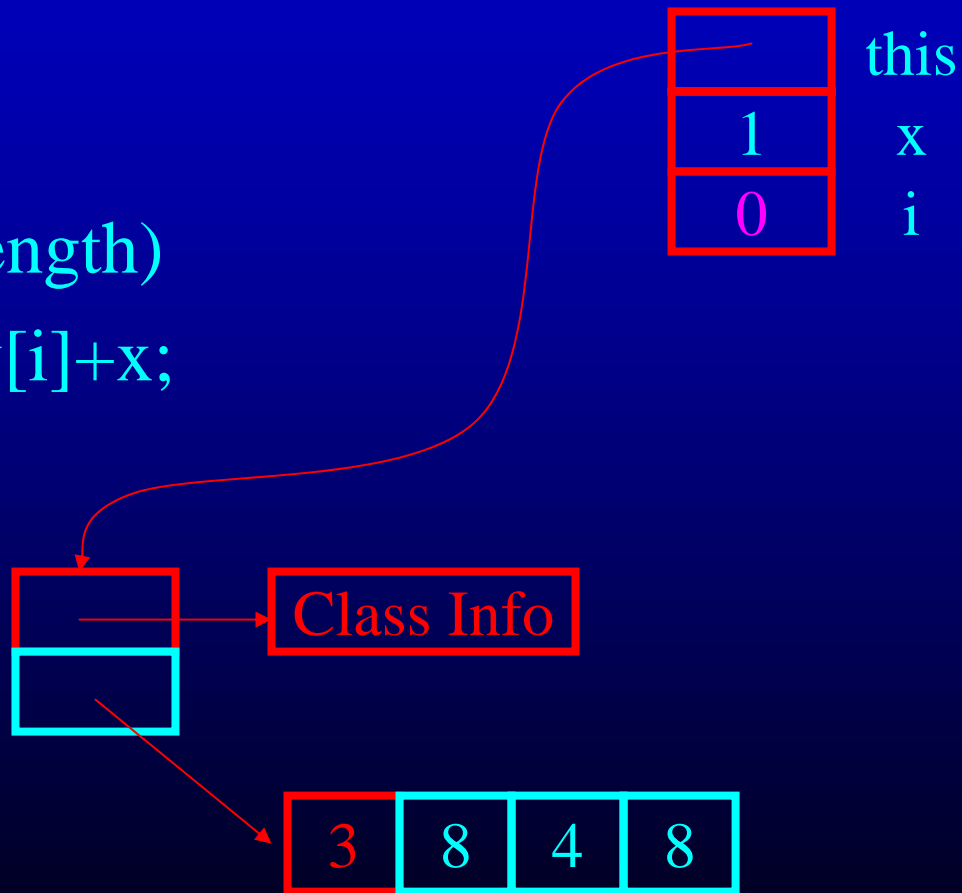
```
    i = 0;
```

```
    while (i < v.length)
```

```
        v[i] = v[i]+x;
```

```
        i = i+1;
```

```
}
```



# Executing Vector Add Method

```
void add(int x) {
```

```
    int i;
```

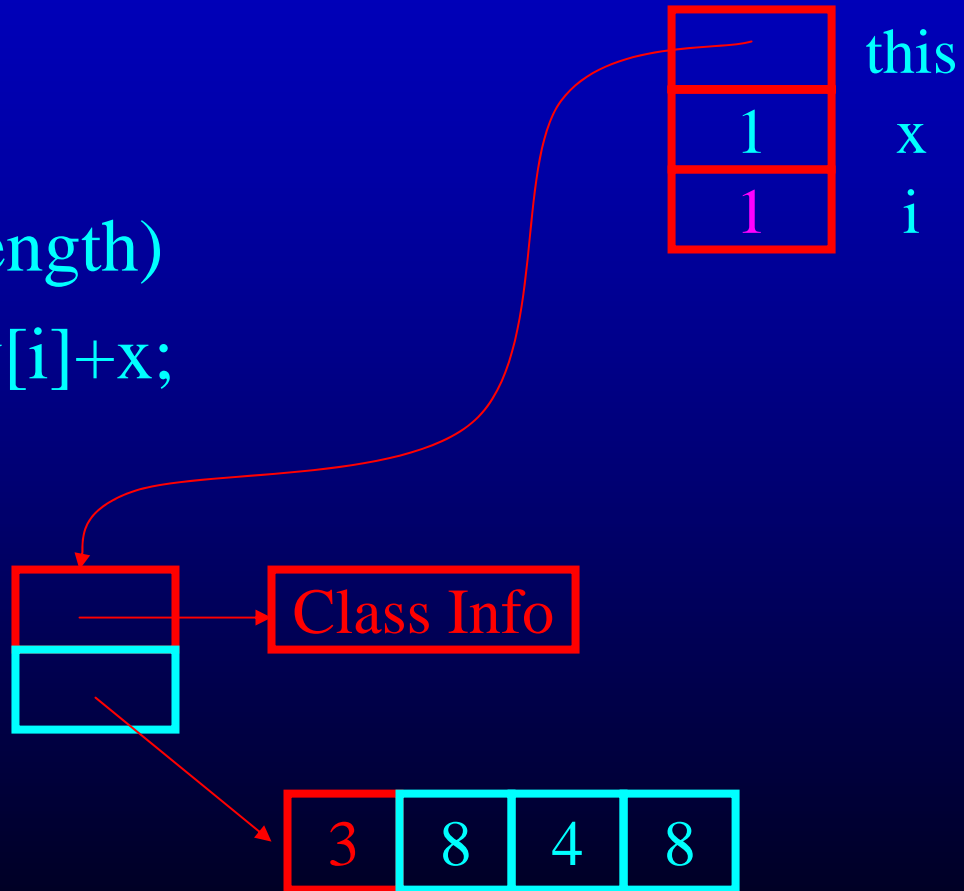
```
    i = 0;
```

```
    while (i < v.length)
```

```
        v[i] = v[i]+x;
```

```
        i = i+1;
```

```
}
```





# Executing Vector Add Method

```
void add(int x) {
```

```
    int i;
```

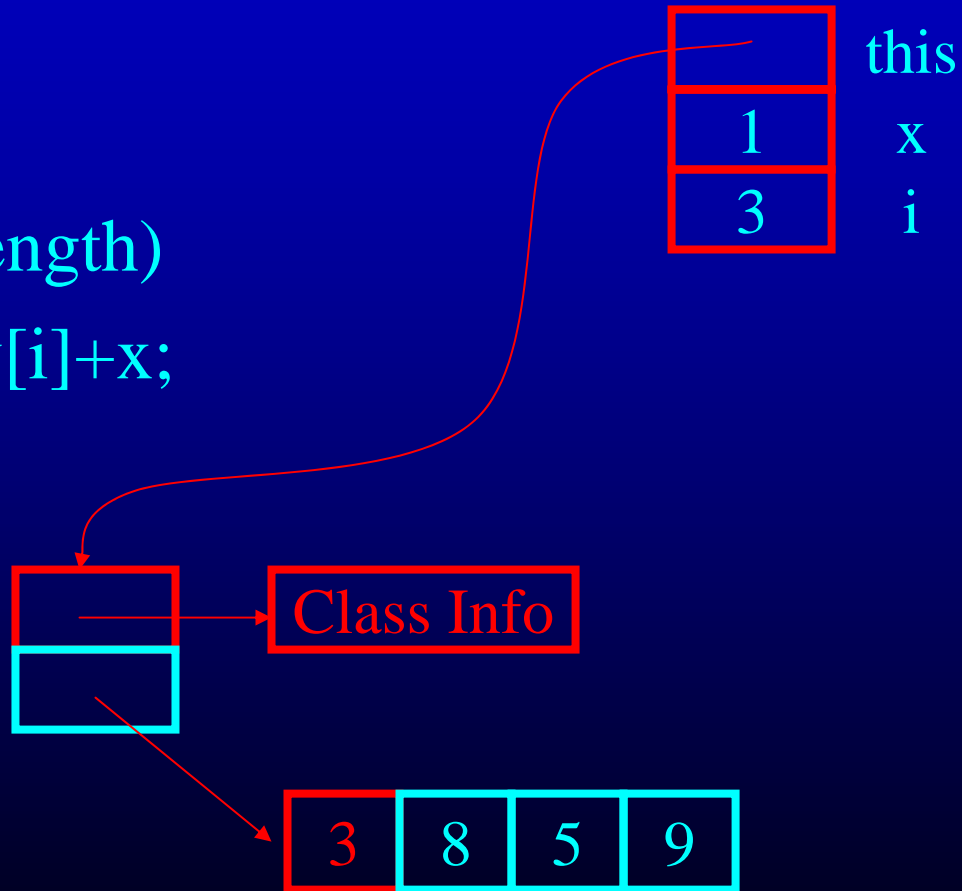
```
    i = 0;
```

```
    while (i < v.length)
```

```
        v[i] = v[i]+x;
```

```
        i = i+1;
```

```
}
```



What does the compiler have to do  
to make all of this work?

# Compilation Tasks

- Determine Format of Objects and Arrays
- Determine Format of Call Stack
- Generate Code to Read Values
  - this, parameters, locals, array elements, object fields
- Generate Code to Evaluate Expressions
- Generate Code to Write Values
- Generate Code for Control Constructs

# Further Complication - Inheritance

## Object Extension

# Inheritance Example - Point Class

```
class point {  
    int c;  
    int getColor() { return(c); }  
    int distance() { return(0); }  
}
```

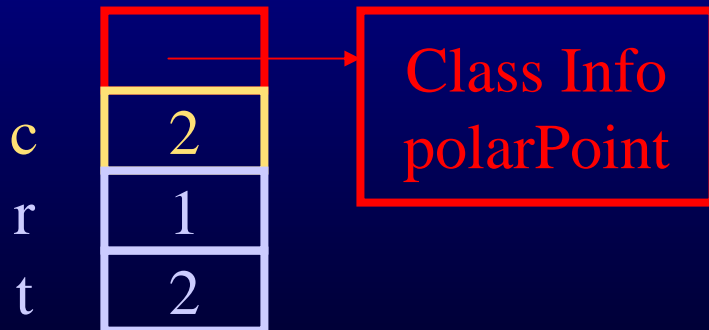
# Point Subclasses

```
class cartesianPoint extends point{  
    int x, y;  
    int distance() { return(x*x + y*y); }  
}
```

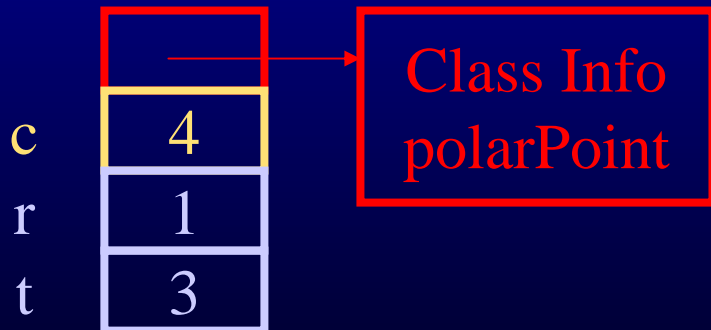
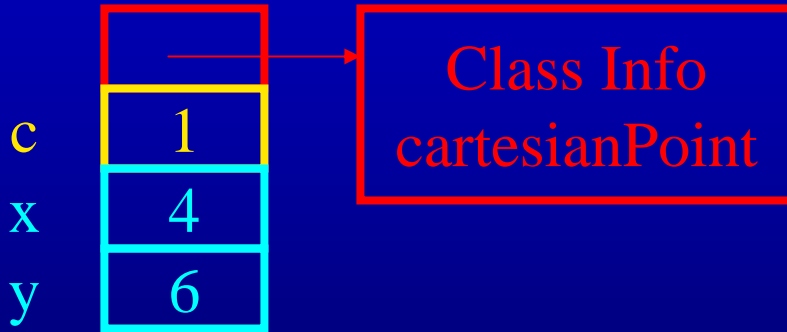
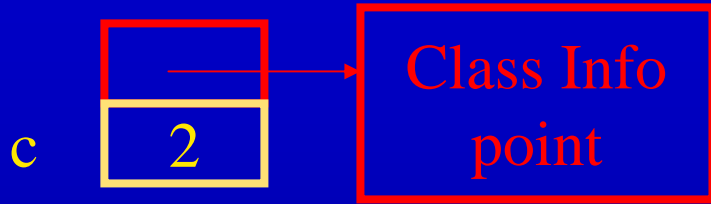
```
class polarPoint extends point {  
    int r, t;  
    int distance() { return(r*r); }  
    int angle() { return(t); }  
}
```

# Implementing Object Fields

- Each object is a contiguous piece of memory
- Fields from inheritance hierarchy allocated sequentially in piece of memory
- Example: polarPoint object



# Point Objects





# Compilation Tasks

- Determine Object Format in Memory
  - Fields from Parent Classes
  - Fields from Current Class
- Generate Code for Methods
  - Field, Local Variable and Parameter Accesses
  - Method Invocations

# Symbol Tables - Key Concept in Compilation

- Compiler Uses Symbol Tables to Produce
  - Object Layout in Memory
  - Code to
    - Access Object Fields
    - Access Local Variables
    - Access Parameters
    - Invoke Methods

# Symbol Tables During Translation

## From Parse Tree to IR

- Symbol Tables Map Identifiers (strings) to Descriptors (information about identifiers)
- Basic Operation: Lookup
  - Given A String, find Descriptor
  - Typical Implementation: Hash Table
- Examples
  - Given a class name, find class descriptor
  - Given variable name, find descriptor
    - local descriptor, parameter descriptor, field descriptor

# Hierarchy In Symbol Tables

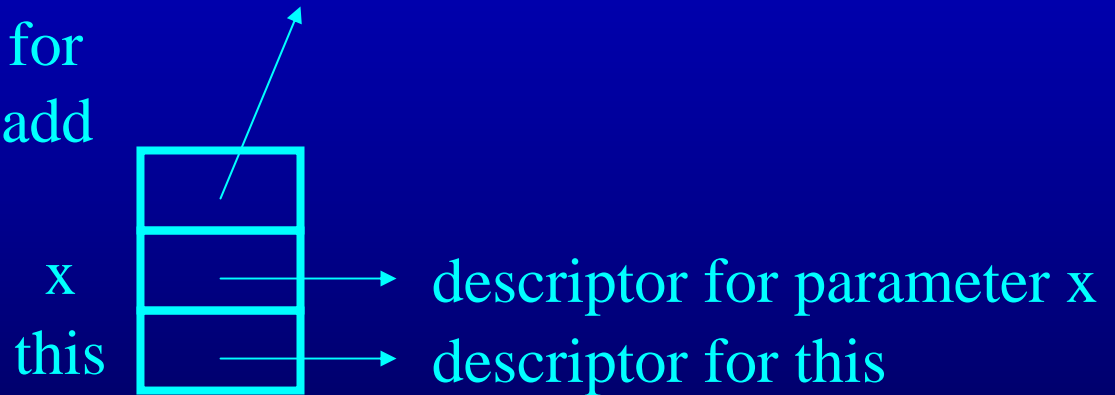
- Hierarchy Comes From
  - Nested Scopes - Local Scope Inside Field Scope
  - Inheritance - Child Class Inside Parent Class
- Symbol Table Hierarchy Reflects These Hierarchies
- Lookup Proceeds Up Hierarchy Until Descriptor is Found

# Hierarchy in vector add Method

Symbol Table for Fields  
of vector Class



Symbol Table for  
Parameters of add

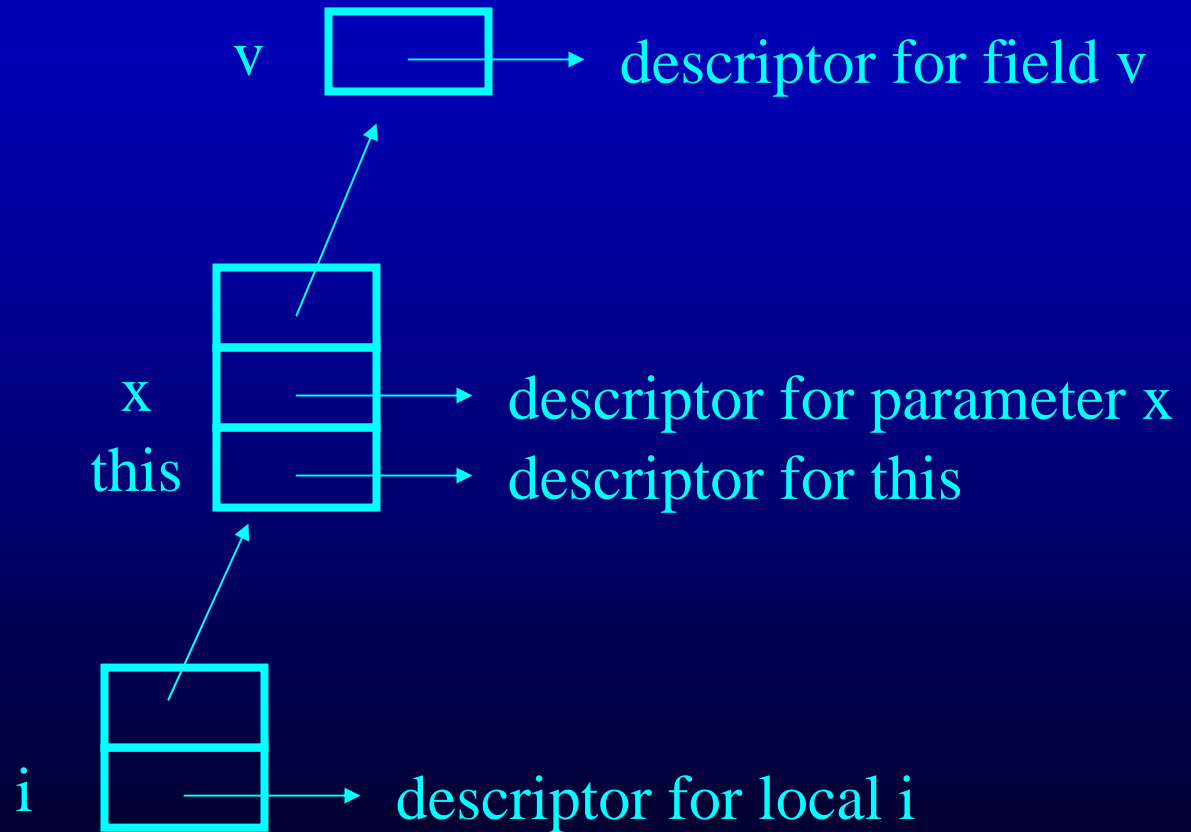


Symbol Table for  
Locals of add



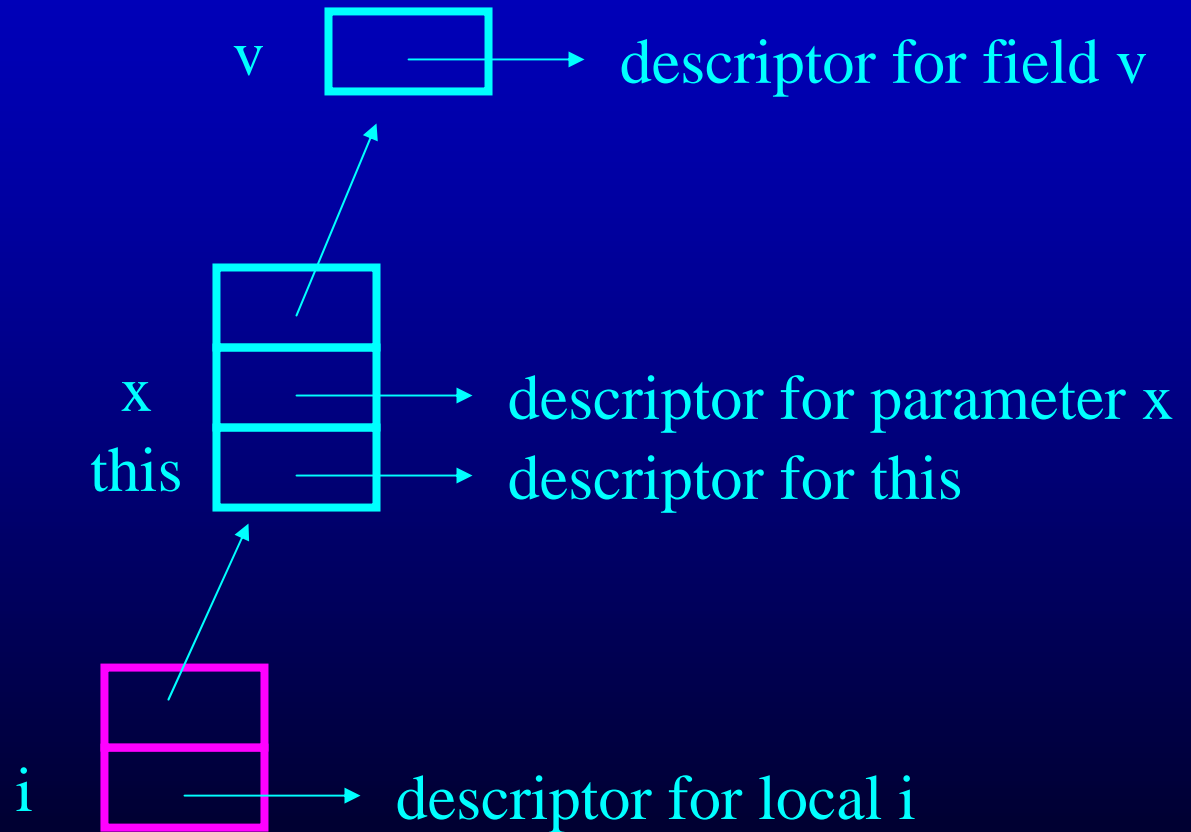
# Lookup In vector Example

- $v[i] = v[i]+x;$



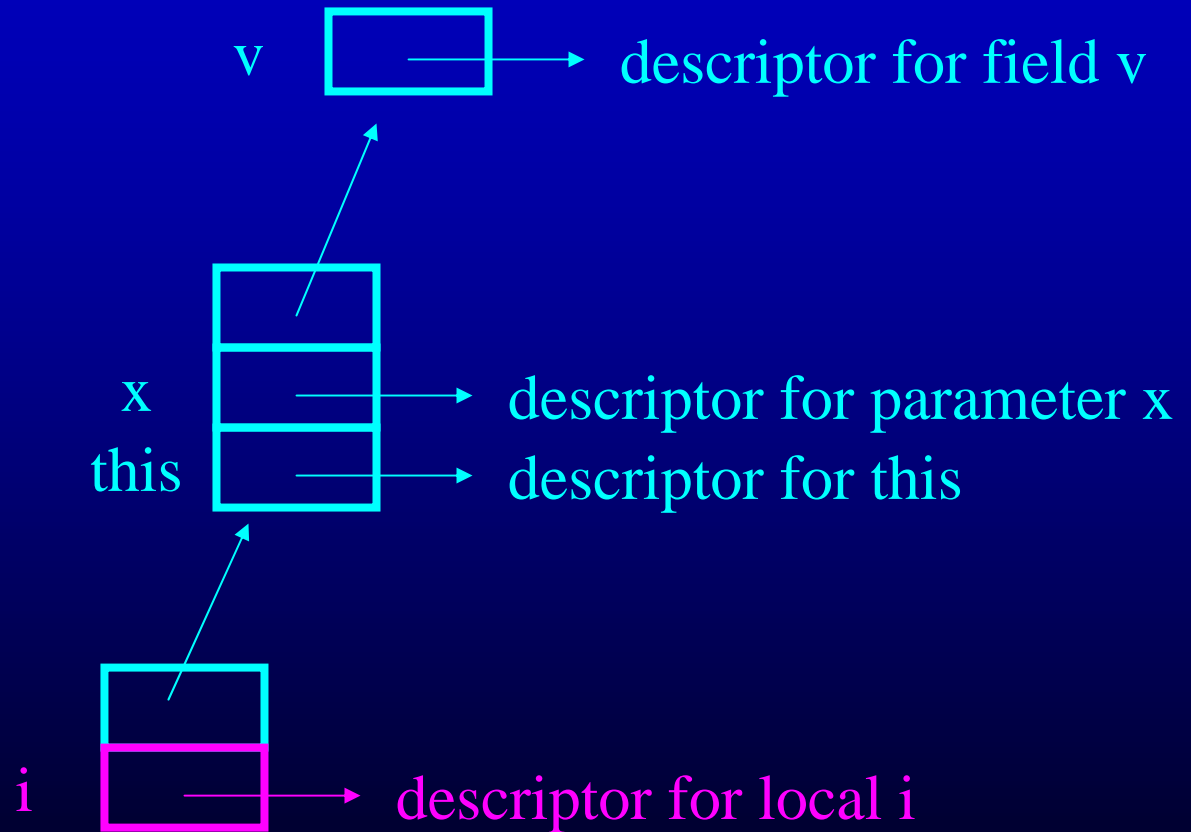
# Lookup i In vector Example

- $v[i] = v[i] + x;$



# Lookup i In vector Example

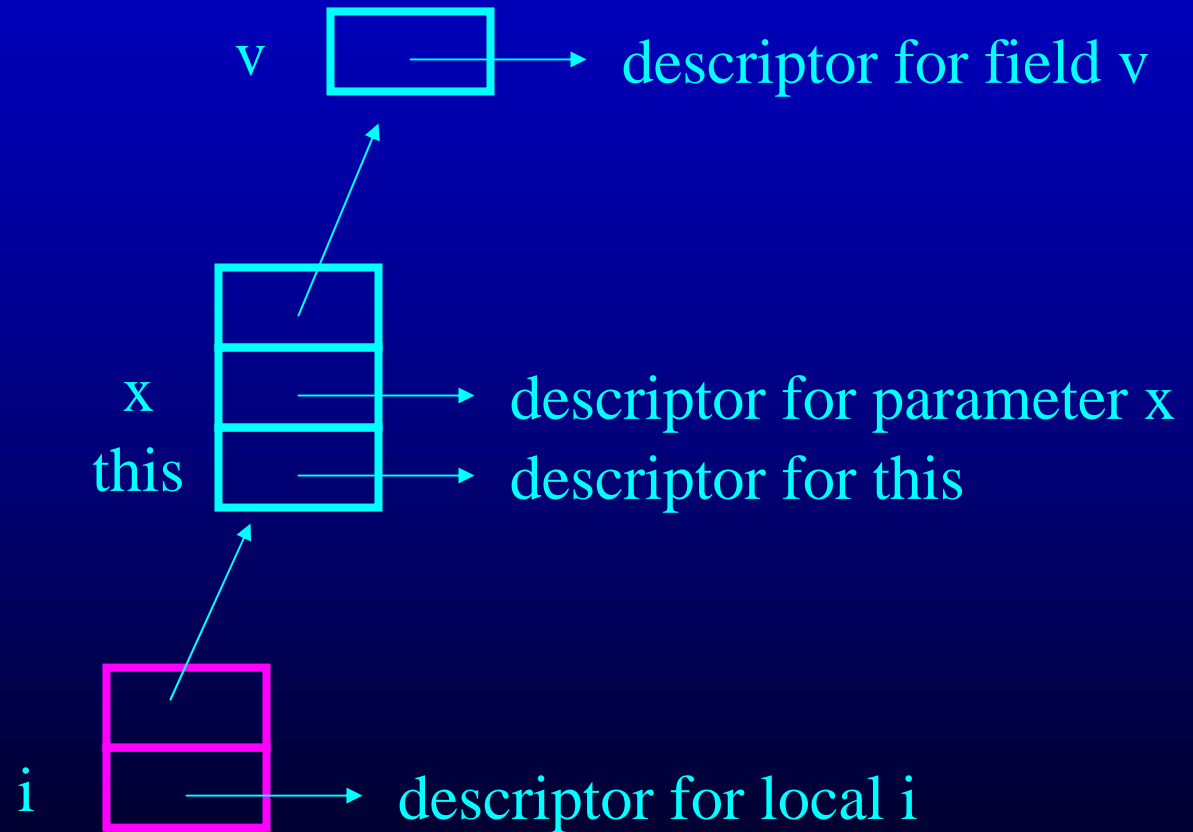
- $v[i] = v[i]+x;$





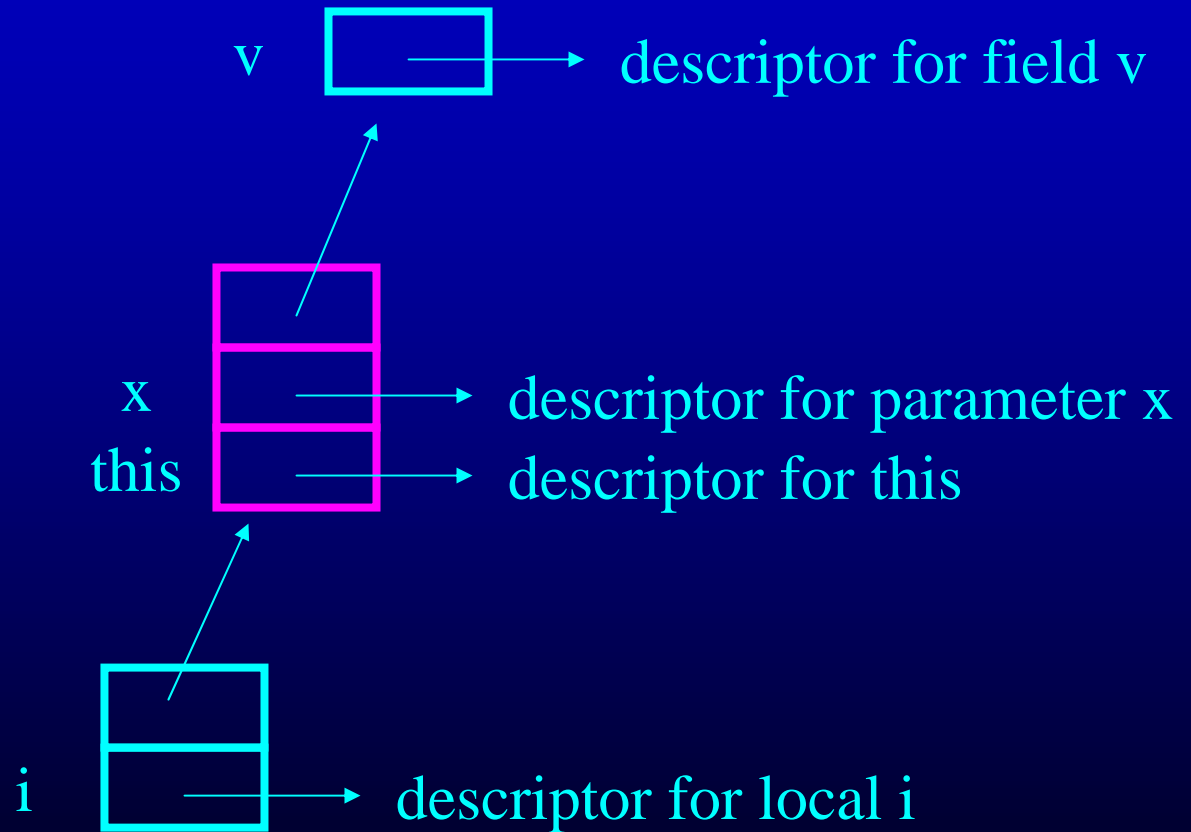
# Lookup x In vector Example

- $v[i] = v[i] + x;$



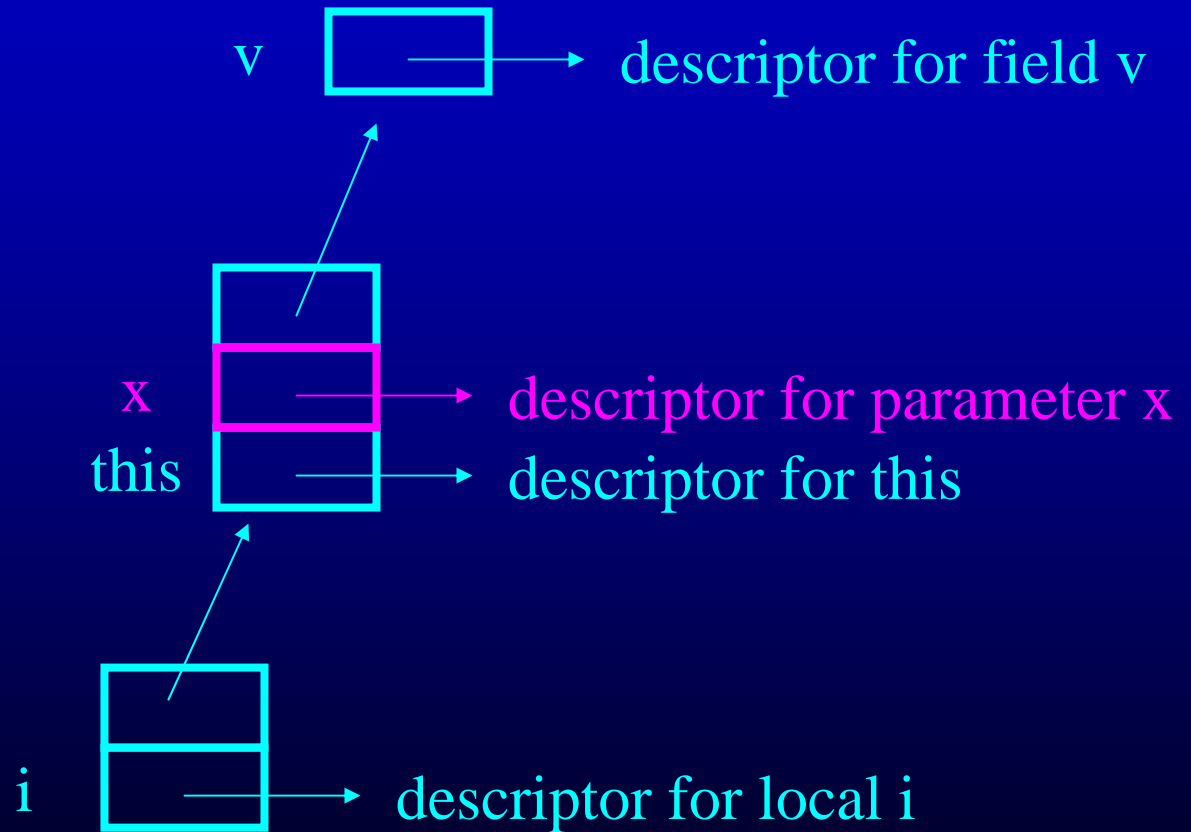
# Lookup x In vector Example

- $v[i] = v[i] + x;$



# Lookup x In vector Example

- $v[i] = v[i] + x;$

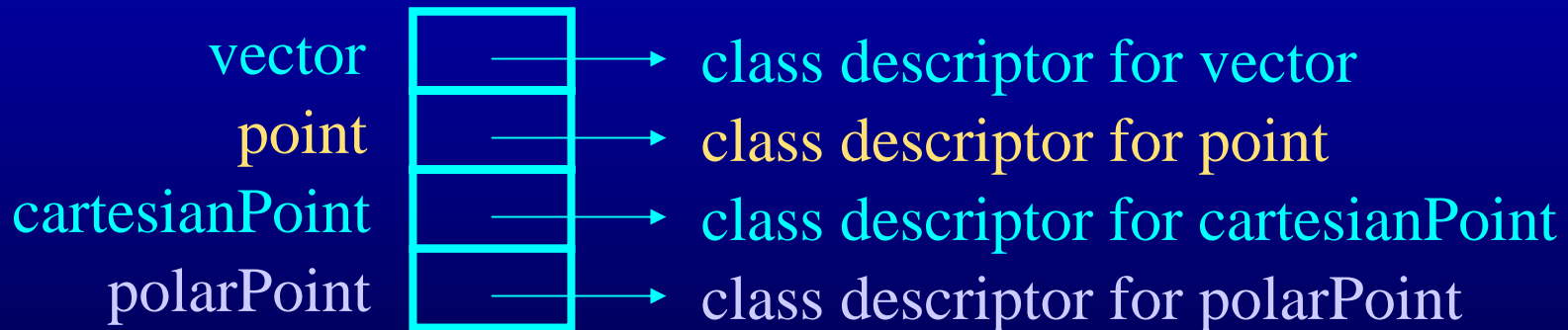


# Descriptors

- What do descriptors contain?
- Information used for code generation and semantic analysis
  - local descriptors - name, type, stack offset
  - field descriptors - name, type, object offset
  - method descriptors
    - signature (type of return value, receiver, and parameters)
    - reference to local symbol table
    - reference to code for method

# Program Symbol Table

- Maps class names to class descriptors
- Typical Implementation: Hash Table

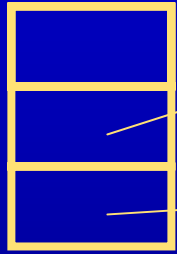


# Class Descriptor

- Has Two Symbol Tables
  - Symbol Table for Methods
    - Parent Symbol Table is Symbol Table for Methods of Parent Class
  - Symbol Table for Fields
    - Parent Symbol Table is Symbol Table for Fields of Parent Class
- Reference to Descriptor of Parent Class

# Class Descriptors for point and cartesianPoint

class descriptor  
for point

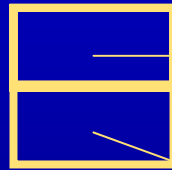


c



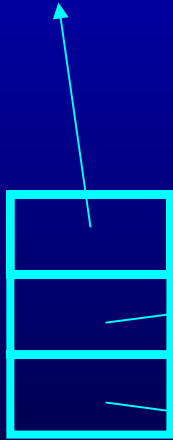
field descriptor for c

getColor  
distance



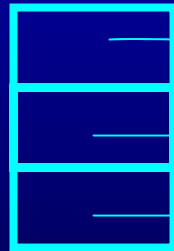
method descriptor  
for getColor

method descriptor  
for distance



x

y



field descriptor for x

field descriptor for y

distance



method descriptor  
for distance

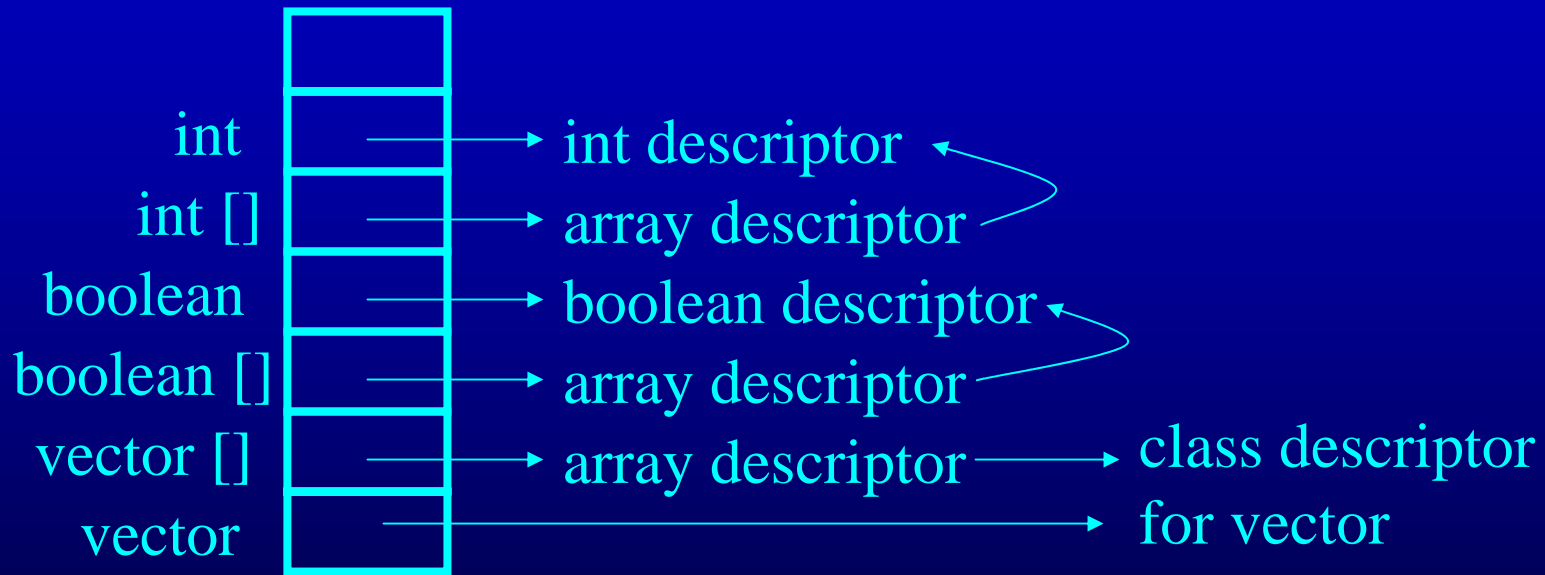
class descriptor  
for cartesianPoint

# Field, Parameter and Local and Type Descriptors

- Field, Parameter and Local Descriptors Refer to Type Descriptors
  - Base type descriptor: int, boolean
  - Array type descriptor, which contains reference to type descriptor for array elements
  - Class descriptor
- Relatively Simple Type Descriptors
- Base Type Descriptors and Array Descriptors Stored in Type Symbol Table



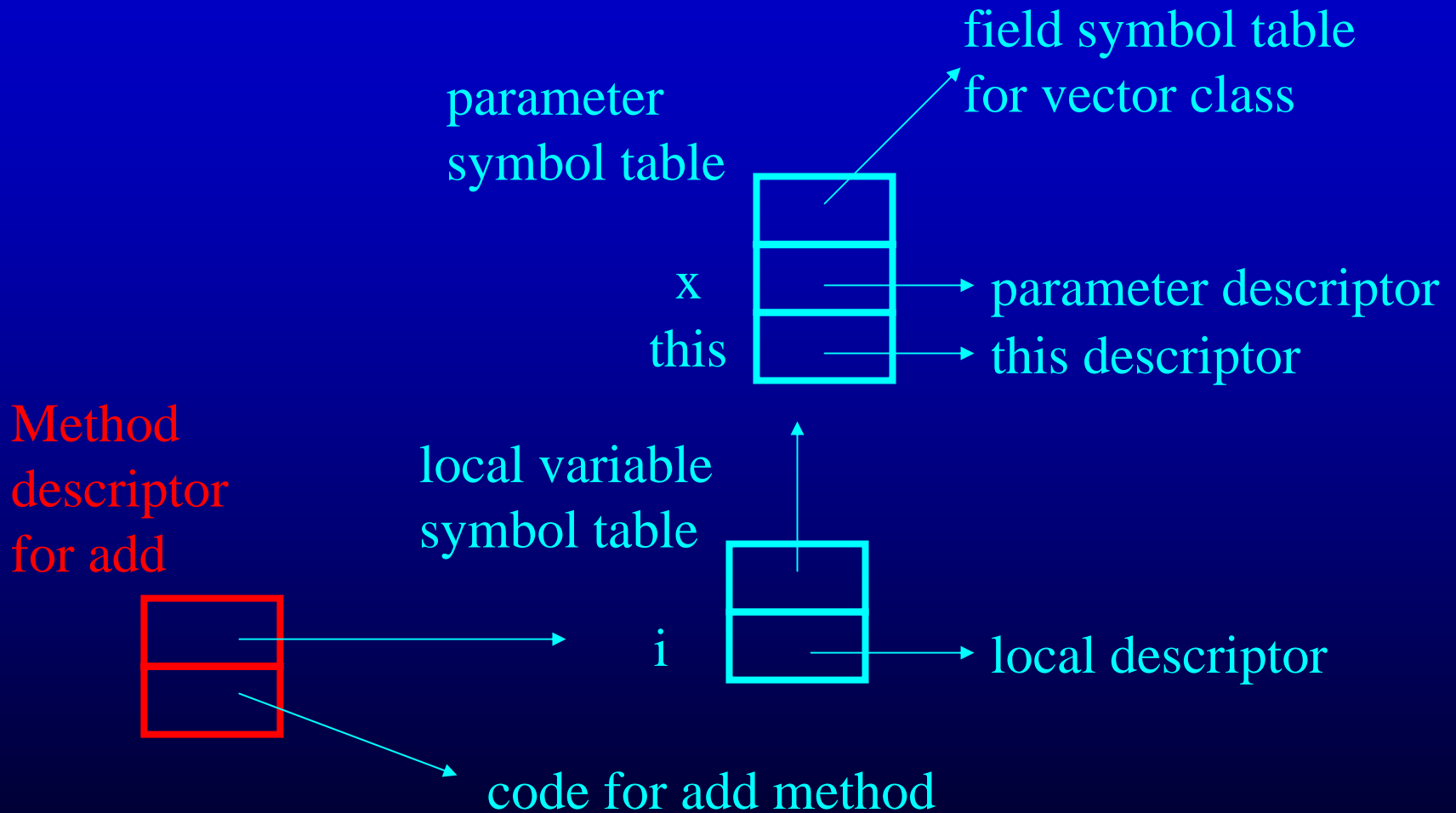
# Example Type Symbol Table



# Method Descriptors

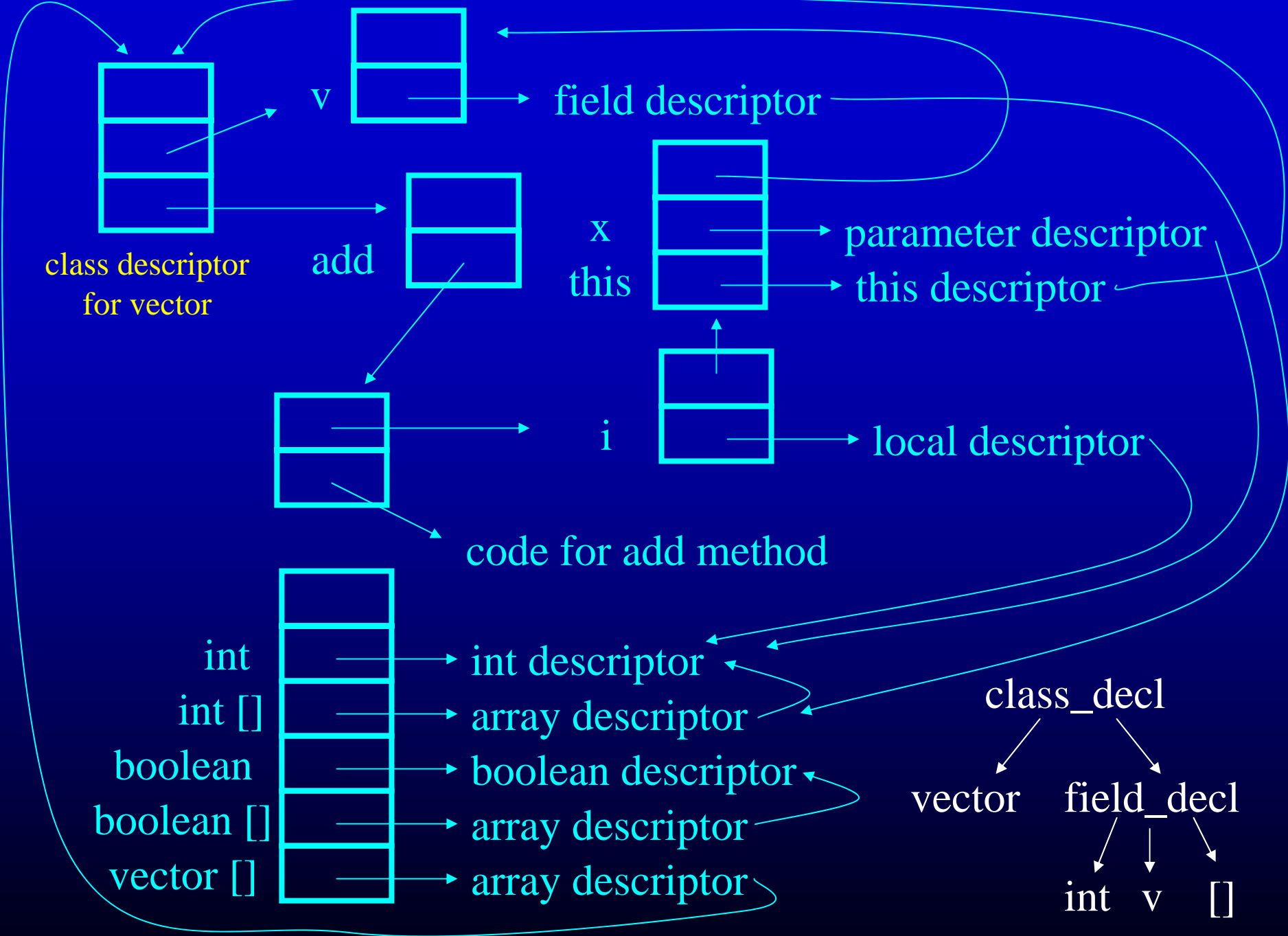
- Contain Reference to Code for Method
- Contain Reference to Local Symbol Table for Local Variables of Method
- Parent Symbol Table of Local Symbol Table is Parameter Symbol Table for Parameters of Method

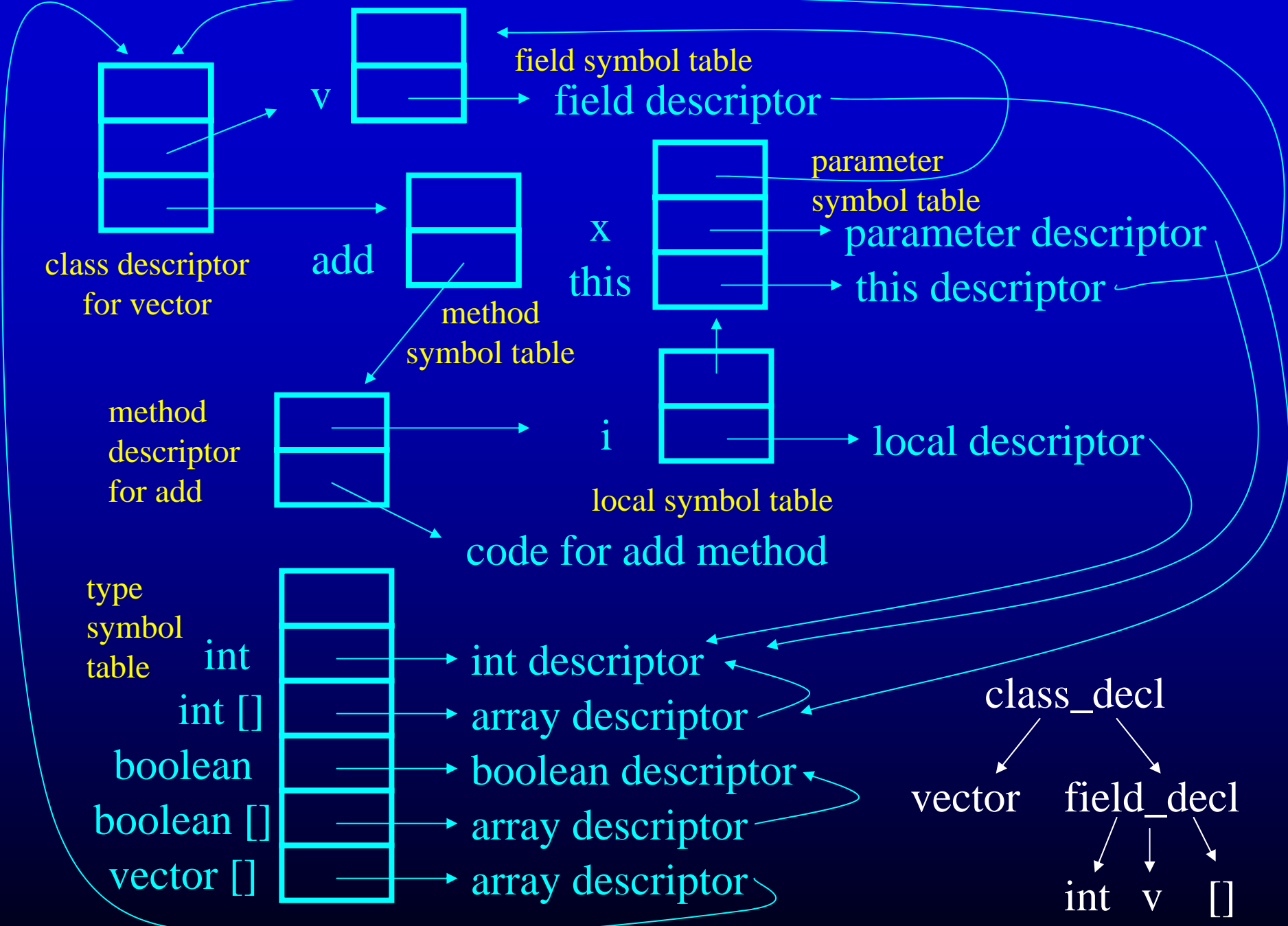
# Method Descriptor for add Method



# Symbol Table Summary

- Program Symbol Table (Class Descriptors)
- Class Descriptors
  - Field Symbol Table (Field Descriptors)
    - Field Symbol Table for SuperClass
  - Method Symbol Table (Method Descriptors)
    - Method Symbol Table for Superclass
- Method Descriptors
  - Local Variable Symbol Table (Local Variable Descriptors)
    - Parameter Symbol Table (Parameter Descriptors)
      - Field Symbol Table of Receiver Class
- Local, Parameter and Field Descriptors
  - Type Descriptors in Type Symbol Table or Class Descriptors

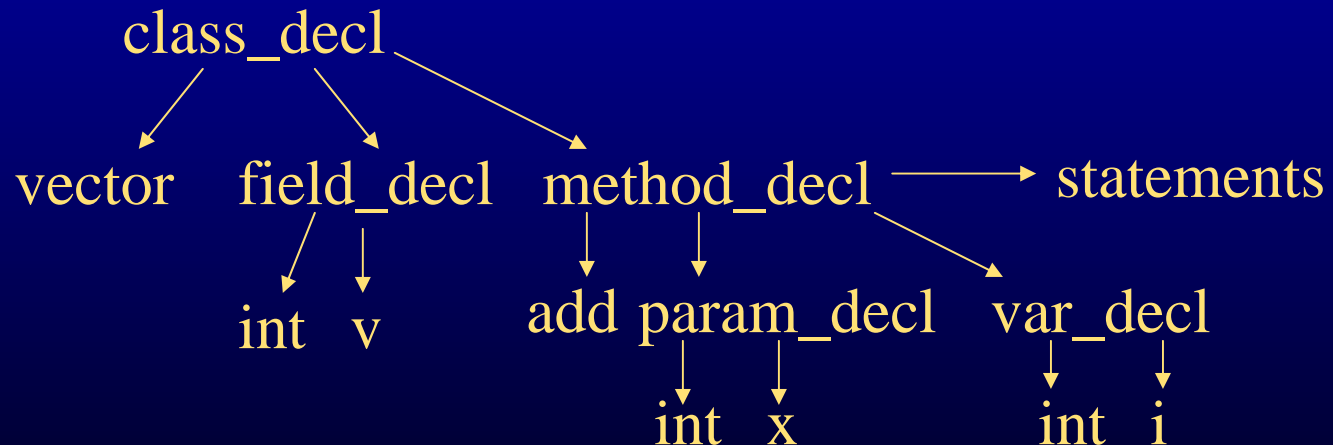




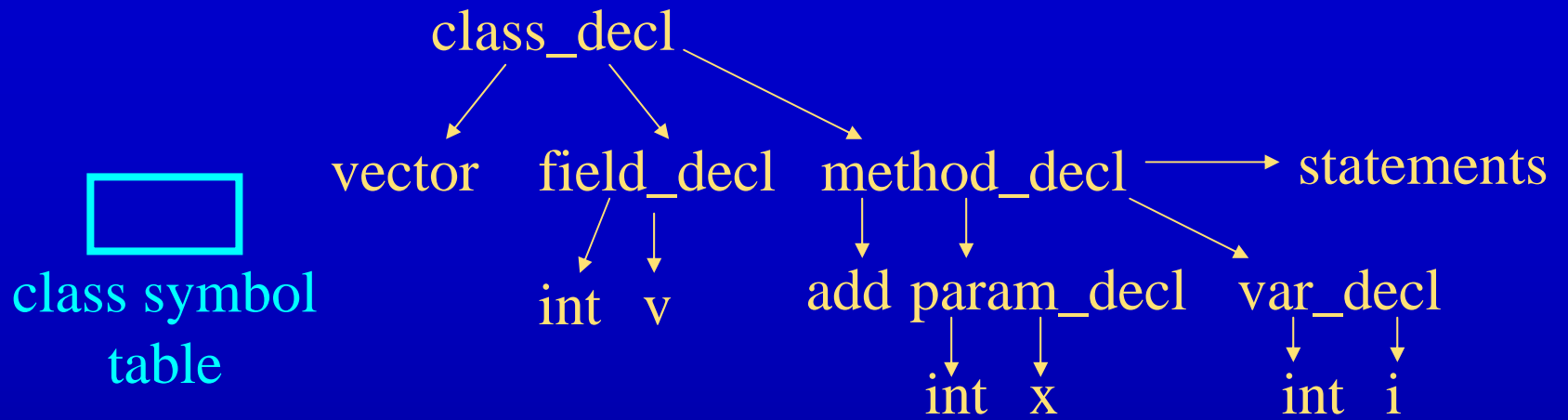
# Translating from Abstract Syntax Trees to Symbol Tables

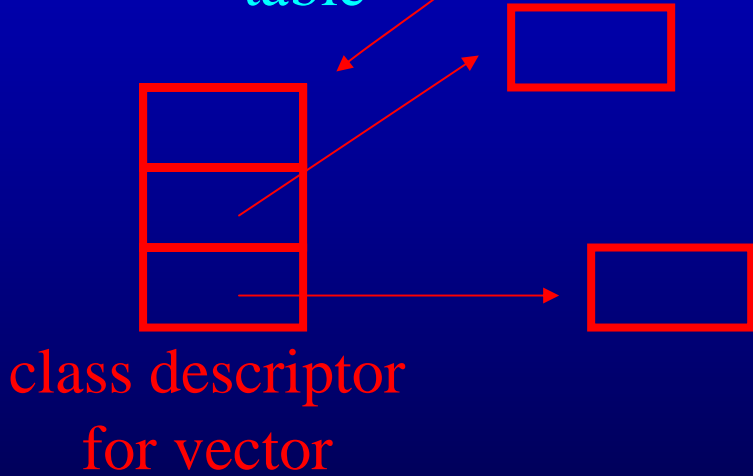
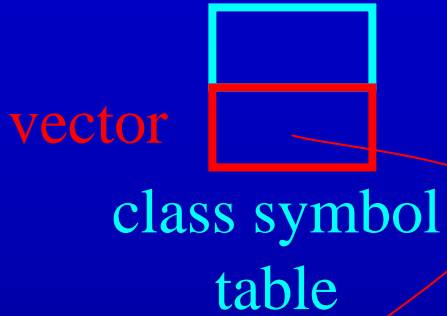
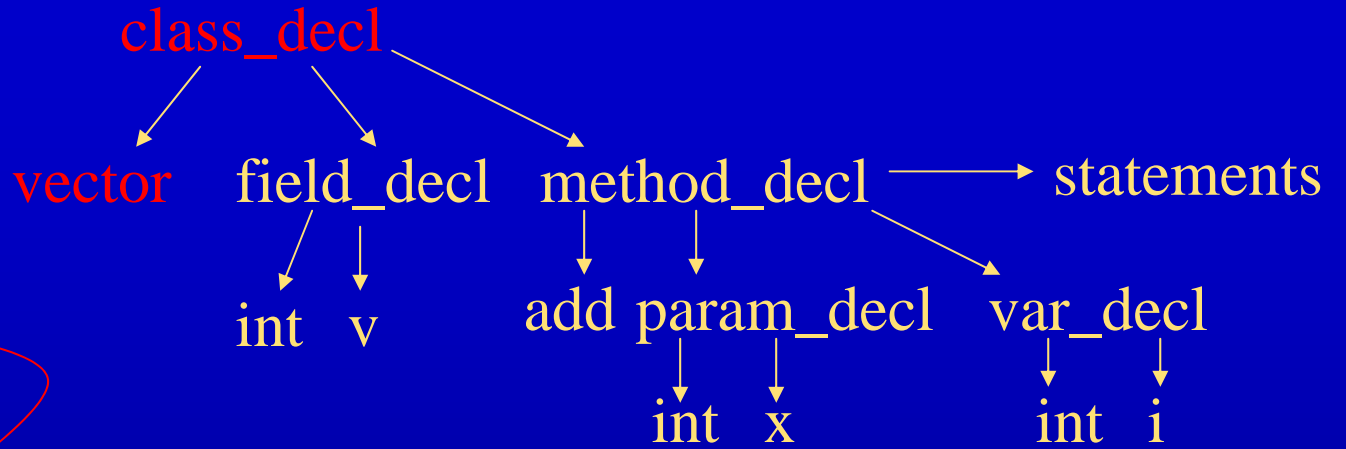
# Example Abstract Syntax Tree

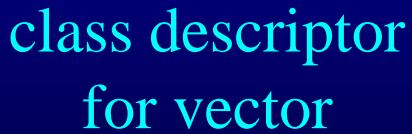
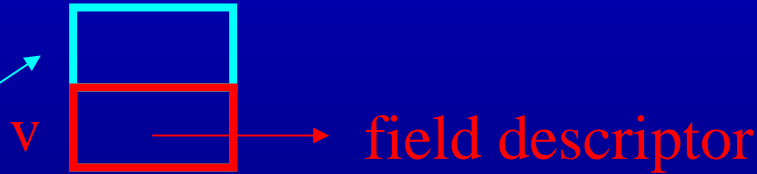
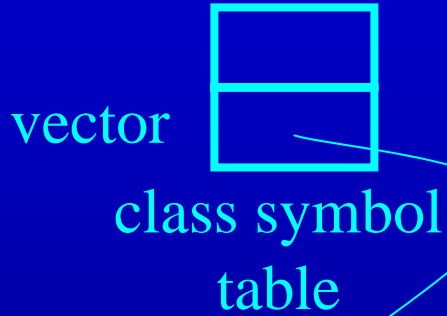
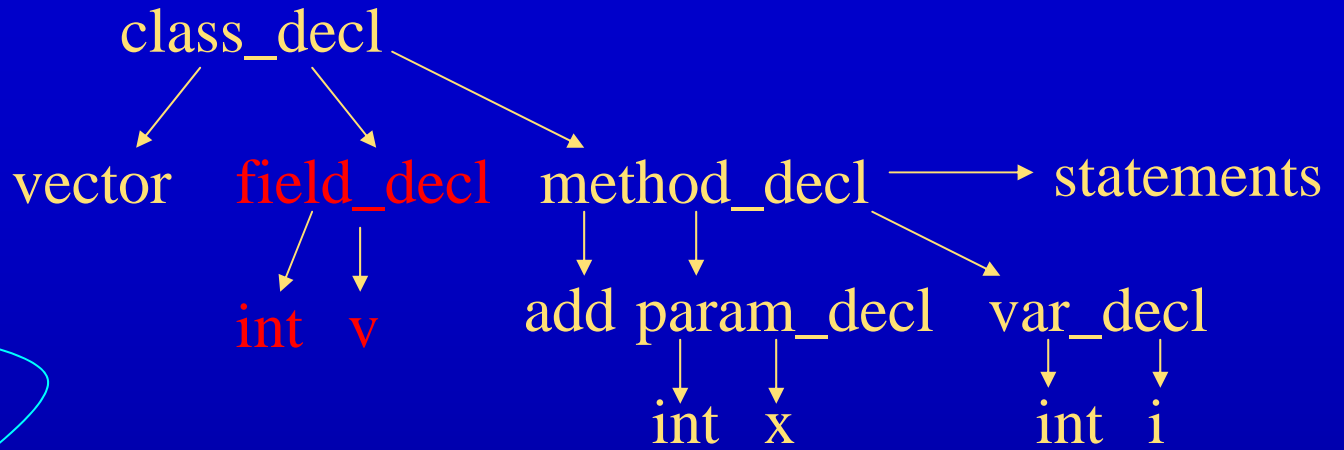
```
class vector {  
  int v[];  
  void add(int x) {  
    int i; i = 0;  
    while (i < v.length) { v[i] = v[i]+x; i = i+1; }  
  }  
}
```

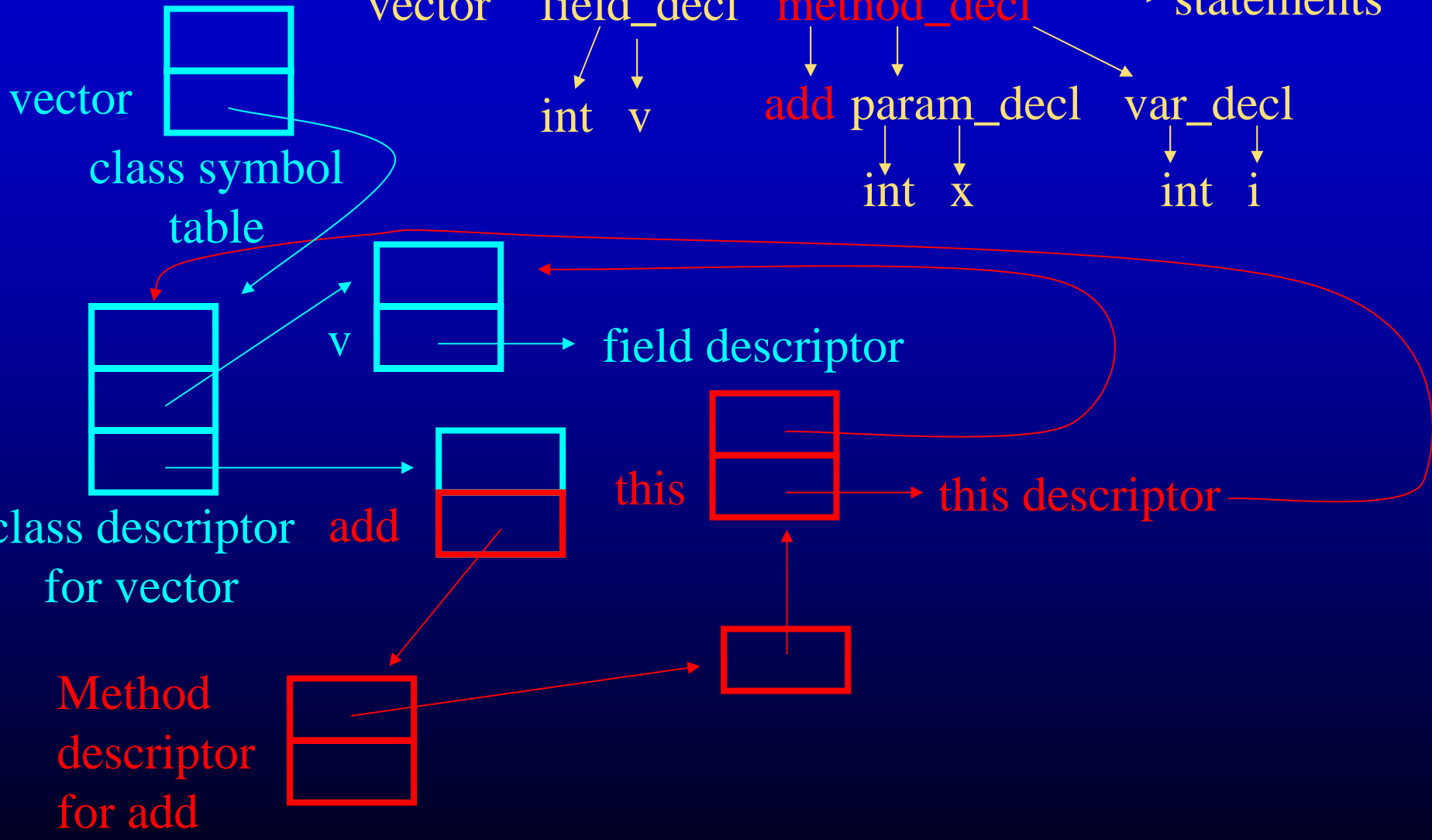
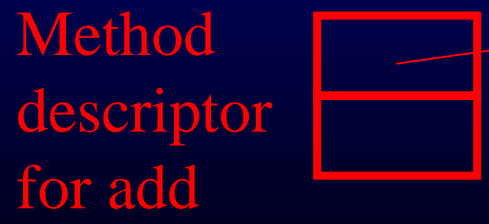
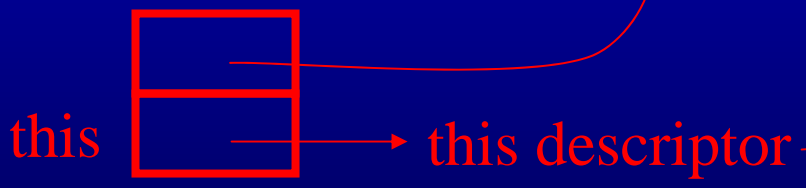
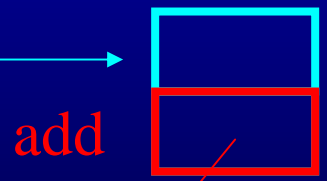
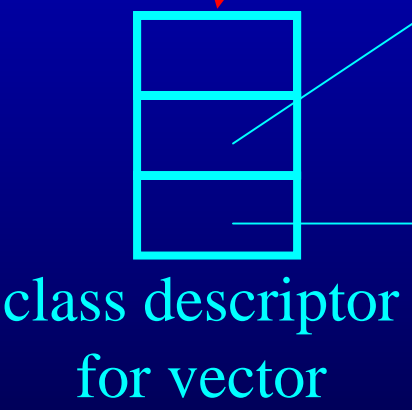
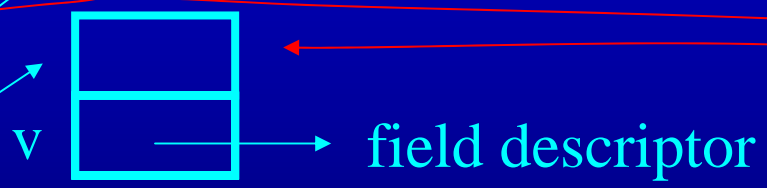
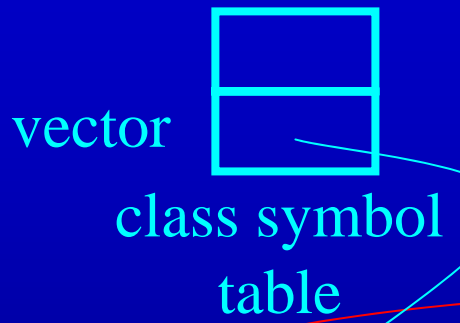
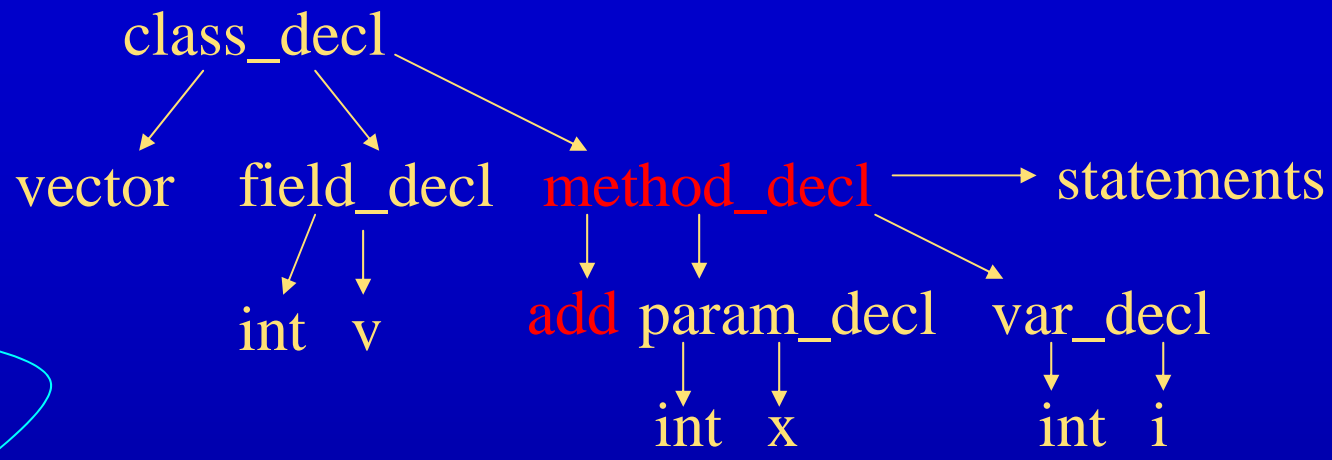


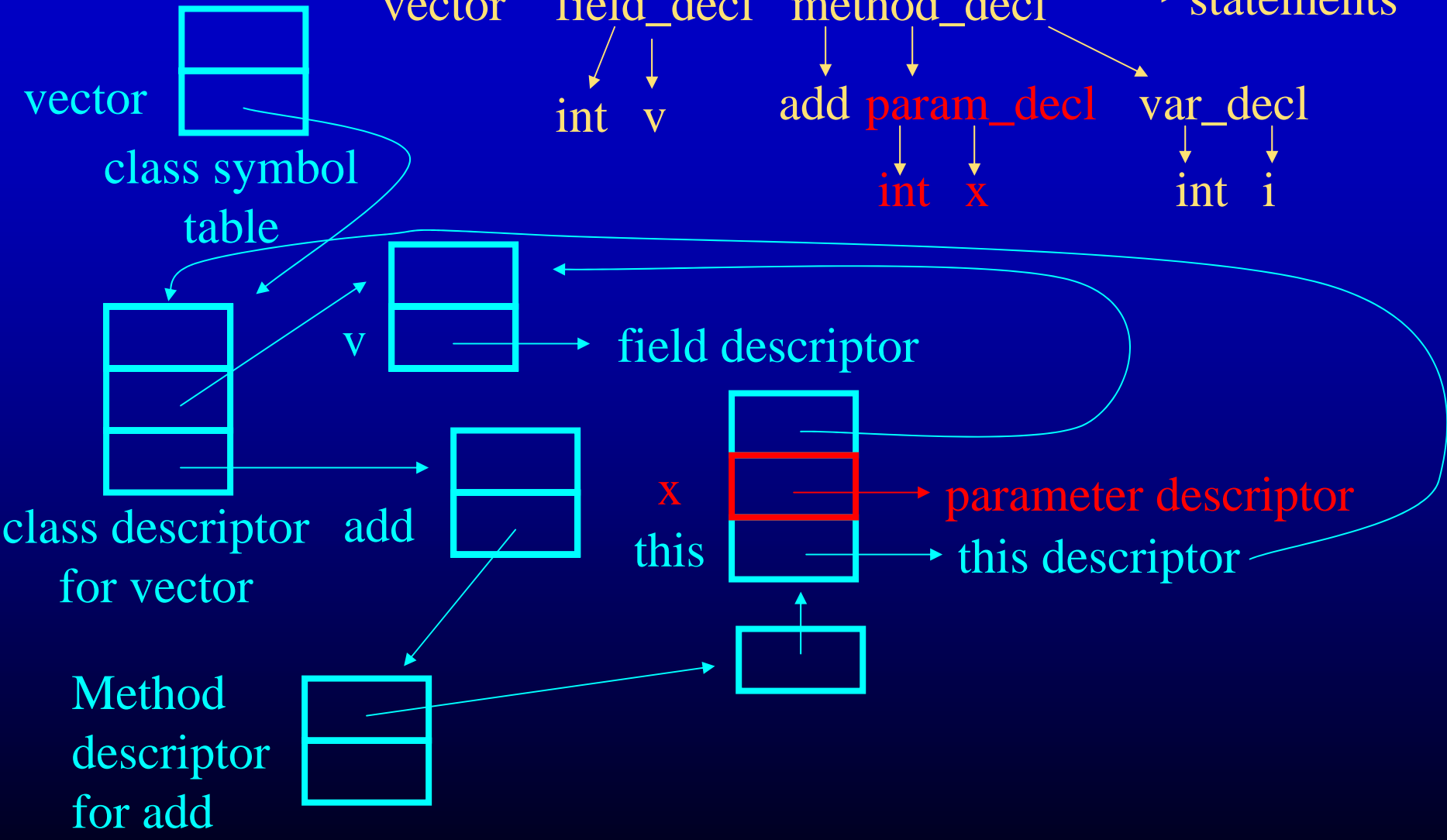
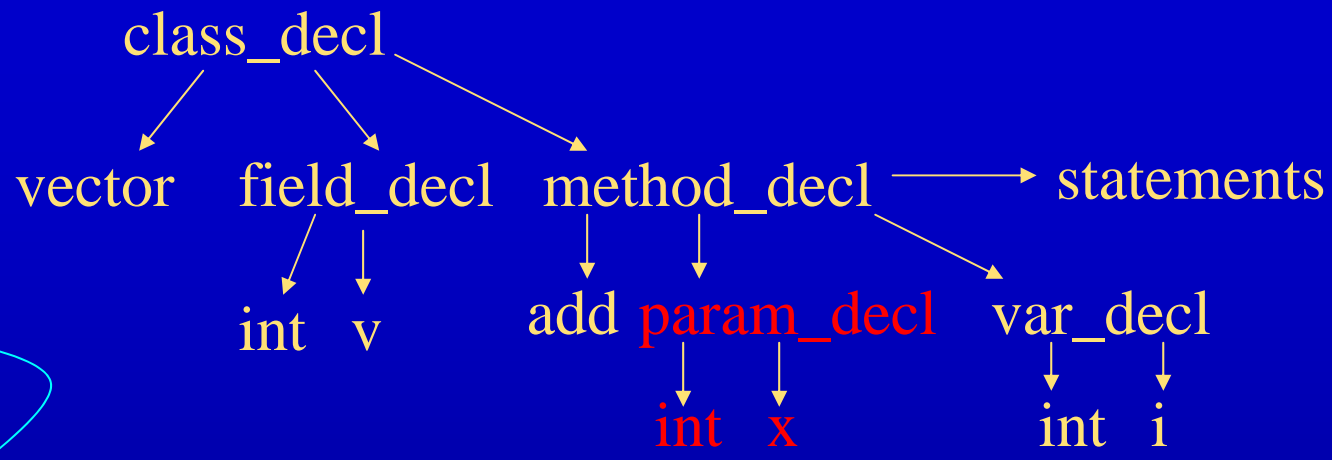


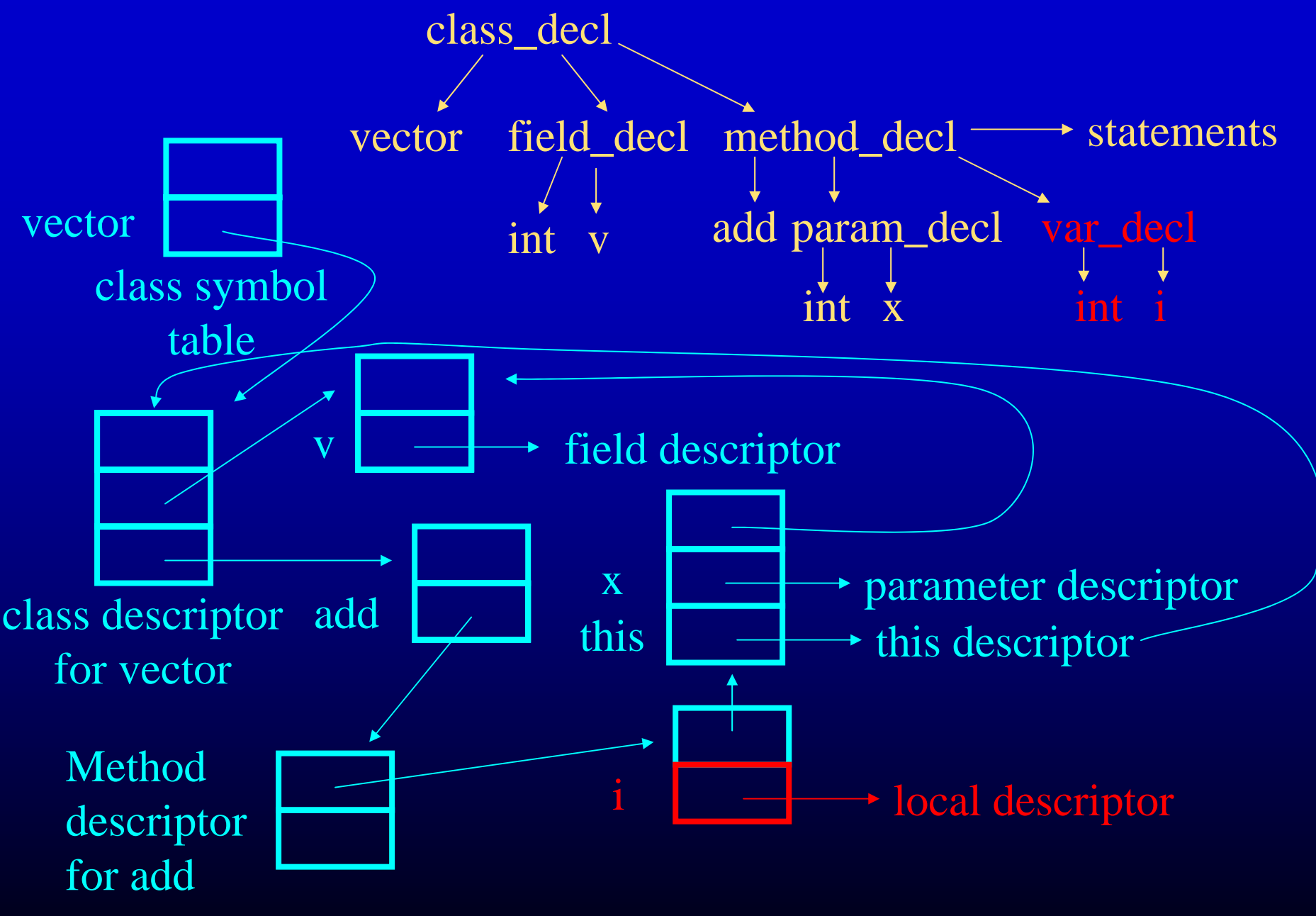












# Representing Code in High-Level Intermediate Representation

# Basic Idea

- Move towards assembly language
- Preserve high-level structure
  - object format
  - structured control flow
  - distinction between parameters, locals and fields
- High-level abstractions of assembly language
  - load and store nodes
  - access abstract locals, parameters and fields, not memory locations directly

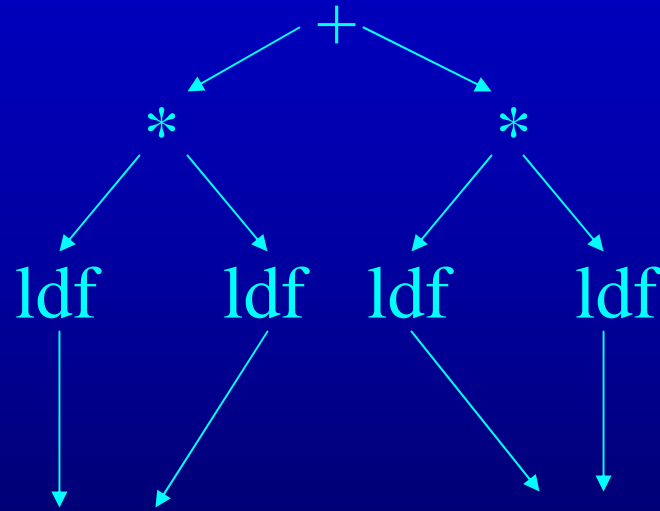


# Representing Expressions

- Expression Trees Represent Expressions
  - Internal Nodes - Operations like +, -, etc.
  - Leaves - Load Nodes Represent Variable Accesses
- Load Nodes
  - ldf node for field accesses - field descriptor
    - (implicitly accesses this - could add a reference to accessed object)
  - ldl node for local variable accesses - local descriptor
  - ldp node for parameter accesses - parameter descriptor
  - lda node for array accesses
    - expression tree for array
    - expression tree for index

# Example

$$x * x + y * y$$

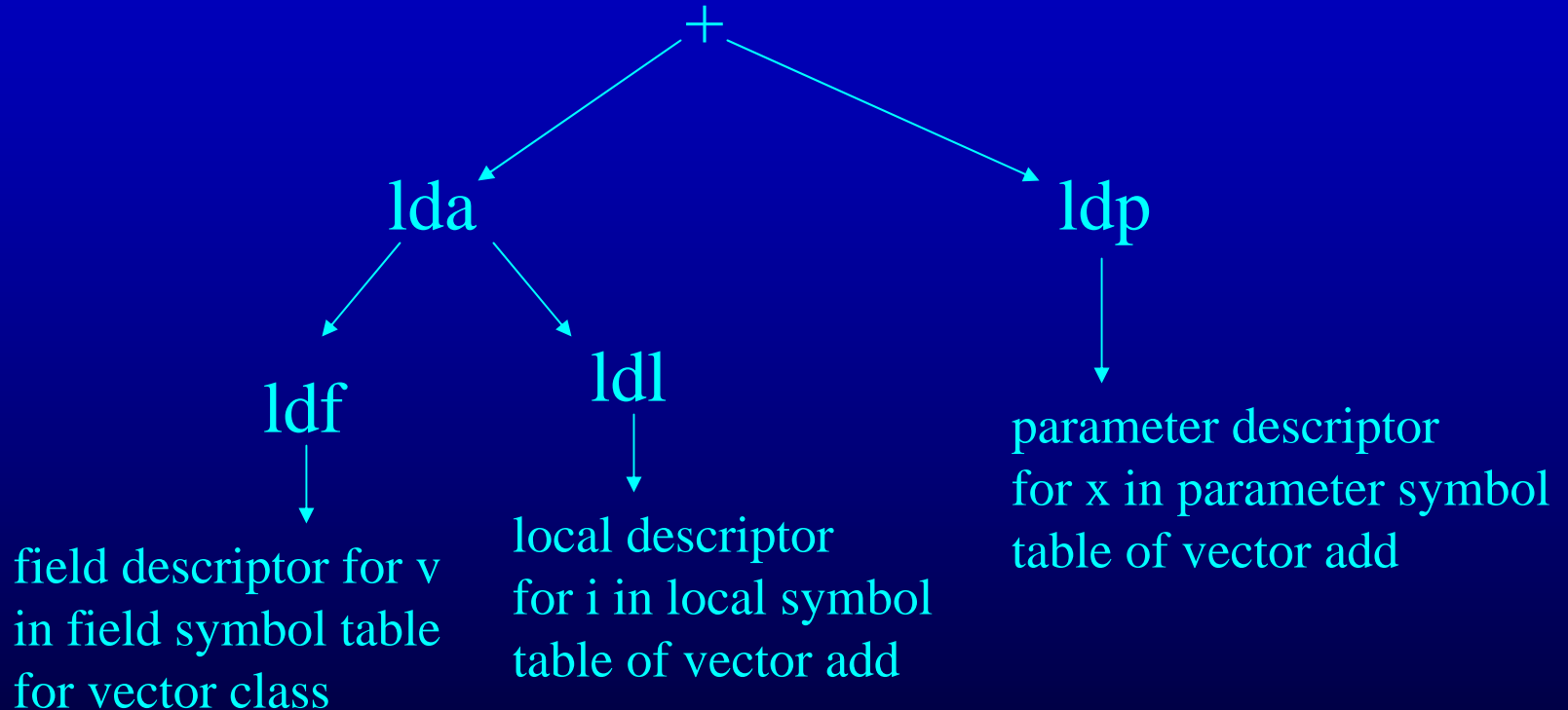


field descriptor for x  
in field symbol table  
for cartesianPoint class

field descriptor for y  
in field symbol table  
for cartesianPoint class

# Example

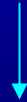
$v[i]+x$



# Special Case: Array Length Operator

- len node represents length of array
  - expression tree for array
- Example: v.length

len



ldf



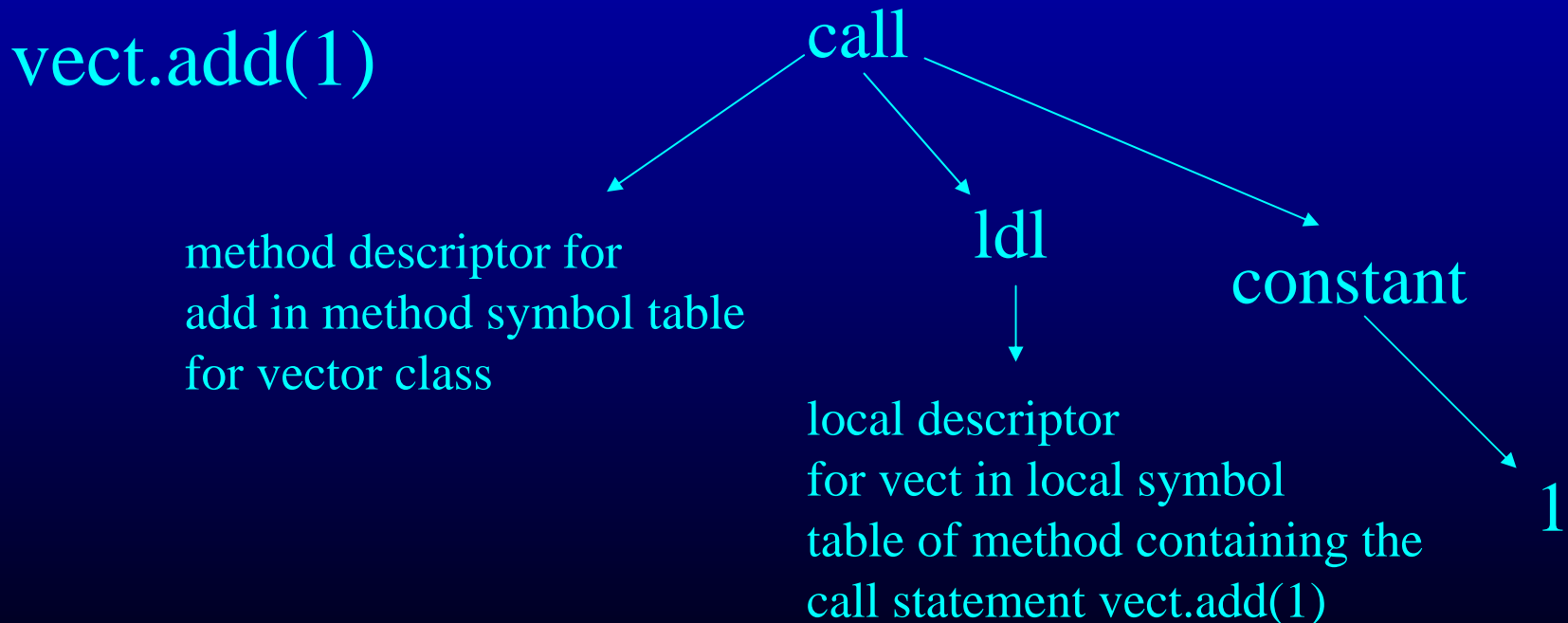
field descriptor for v  
in field symbol table  
for vector class

# Representing Assignment Statements

- Store Nodes
  - stf for stores to fields
    - field descriptor
    - expression tree for stored value
  - stl for stores to local variables
    - local descriptor
    - expression tree for stored value
  - sta for stores to array elements
    - expression tree for array
    - expression tree for index
    - expression tree for stored value

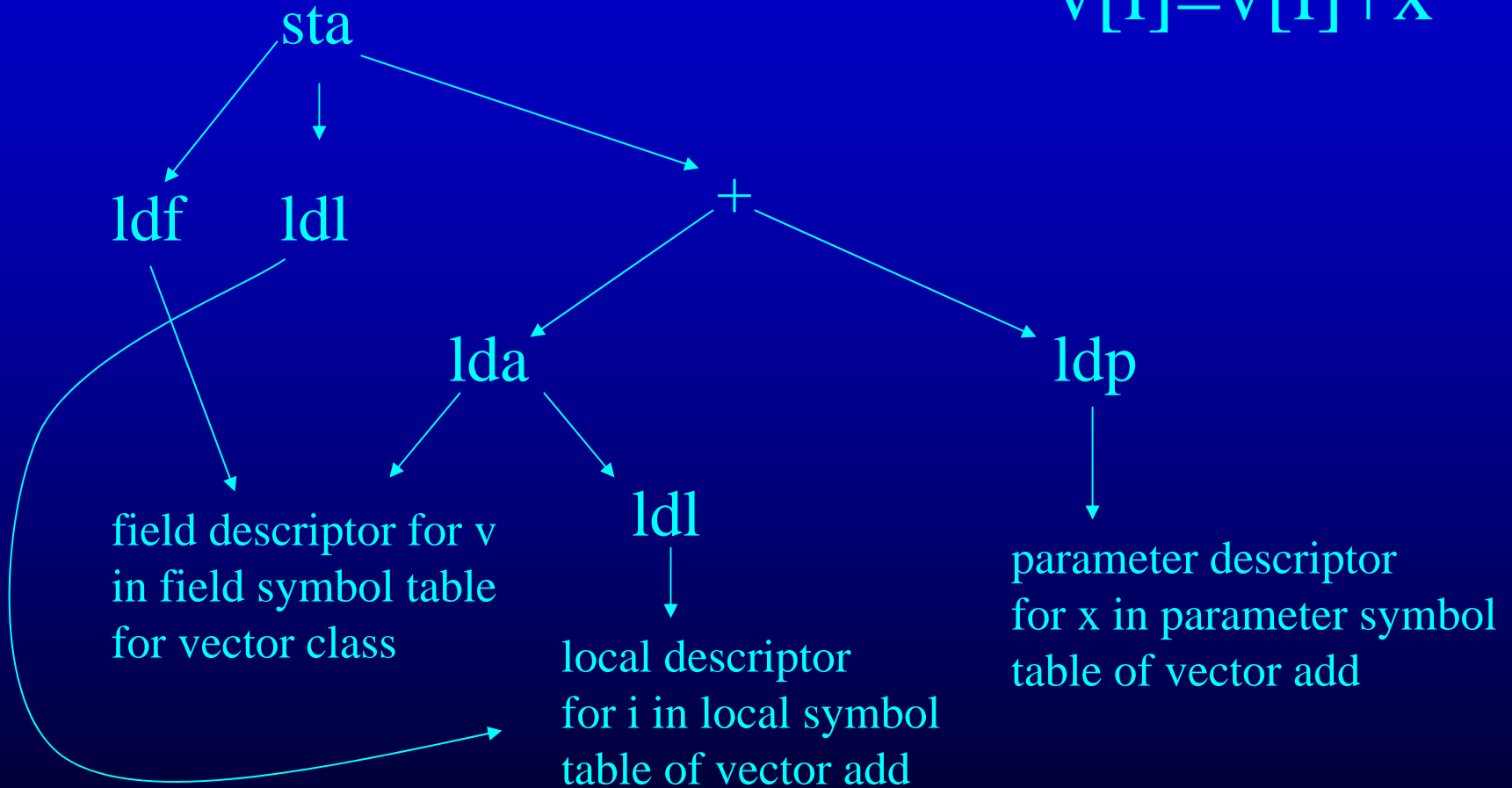
# Representing Procedure Calls

- Call statement
- Refers to method descriptor for invoked method
- Has list of parameters (this is first parameter)



# Example

$$v[i]=v[i]+x$$



# Representing Flow of Control

- Statement Nodes

- sequence node - first statement, next statement

- if node

- expression tree for condition

- then statement node and else statement node

- while node

- expression tree for condition

- statement node for loop body

- return node

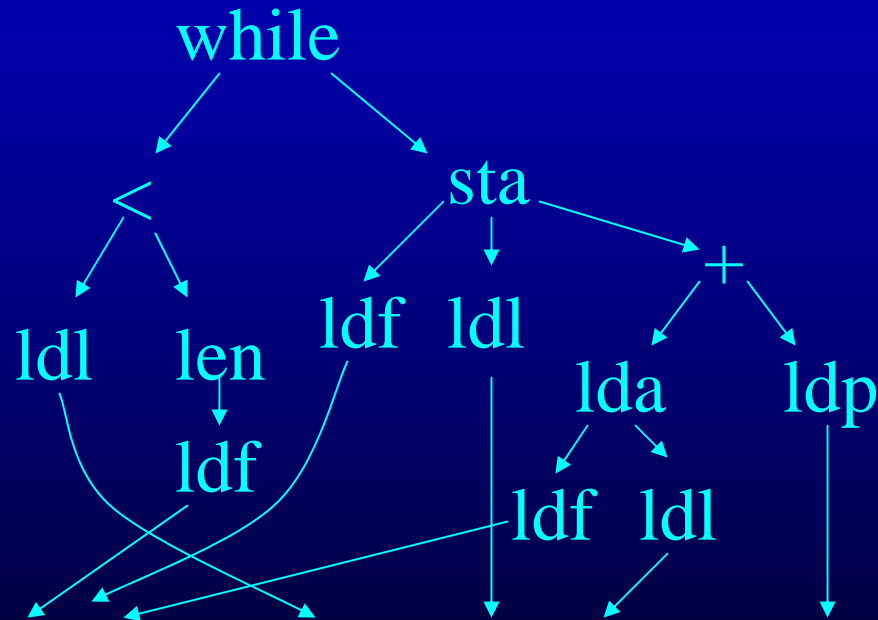
- expression tree for return value



# Example

while (i < v.length)

v[i] = v[i]+x;



field descriptor for v    local descriptor for i    parameter descriptor for x

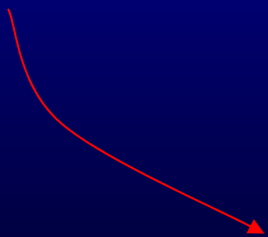
# From Abstract Syntax Trees to Intermediate Representation

```
while (i < v.length)
    v[i] = v[i]+x;
```

field descriptor for v   local descriptor for i   parameter descriptor for x

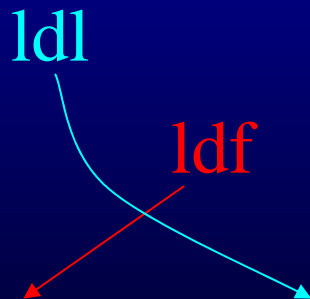
```
while (i < v.length)
    v[i] = v[i]+x;
```

ldl



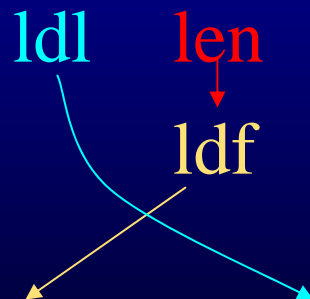
field descriptor for v   local descriptor for i   parameter descriptor for x

```
while (i < v.length)
    v[i] = v[i]+x;
```



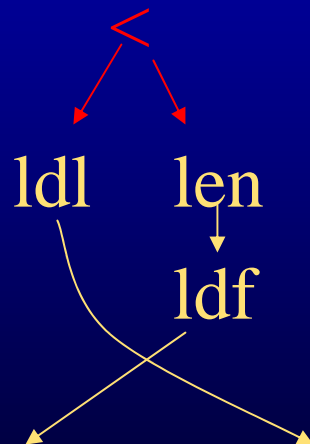
field descriptor for v   local descriptor for i   parameter descriptor for x

```
while (i < v.length)
    v[i] = v[i]+x;
```



field descriptor for v   local descriptor for i   parameter descriptor for x

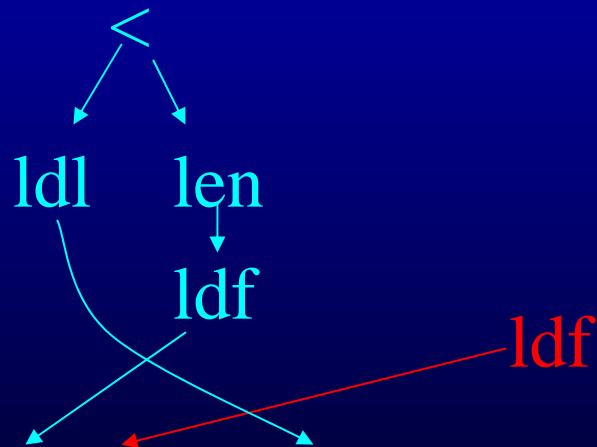
```
while (i < v.length)
    v[i] = v[i]+x;
```



field descriptor for v   local descriptor for i   parameter descriptor for x

```
while (i < v.length)
```

```
  v[i] = v[i]+x;
```

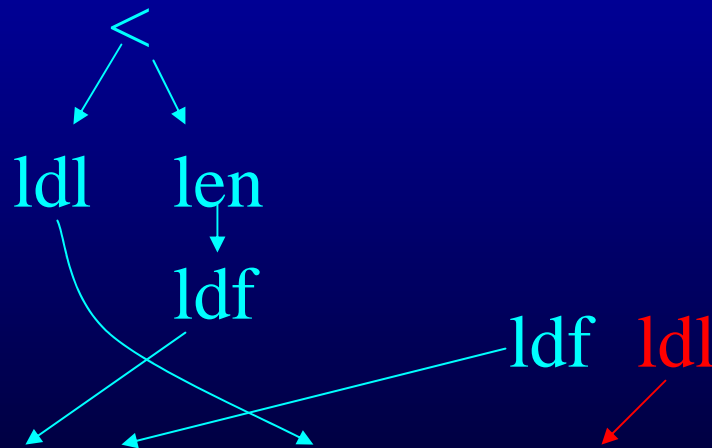


field descriptor for v    local descriptor for i    parameter descriptor for x



```
while (i < v.length)
```

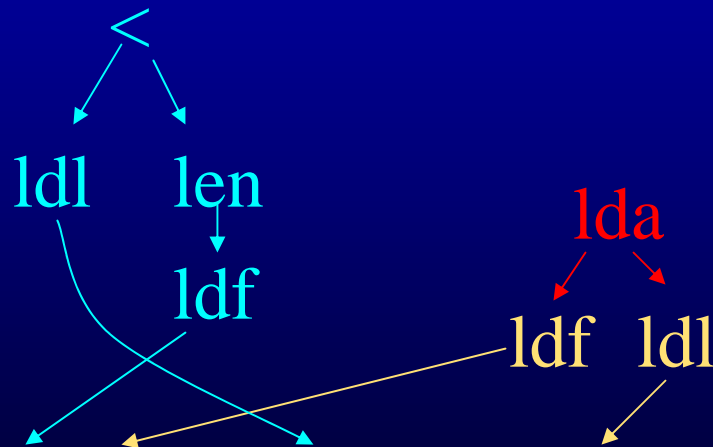
```
  v[i] = v[i]+x;
```



field descriptor for v   local descriptor for i   parameter descriptor for x

while (i < v.length)

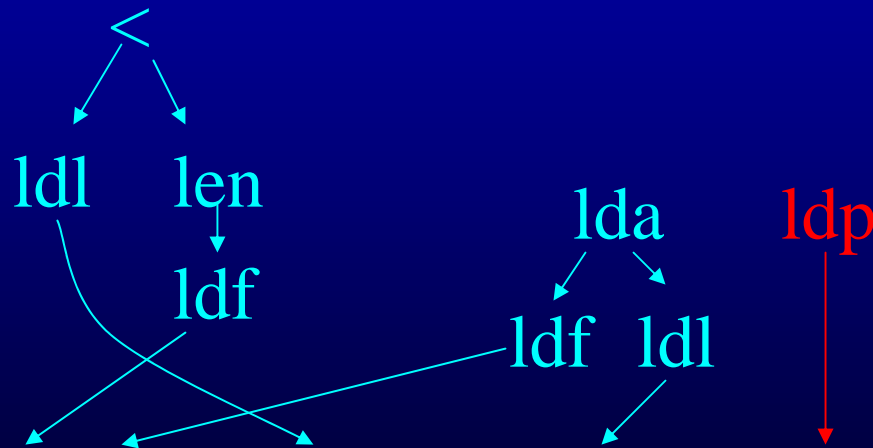
v[i] = v[i]+x;



field descriptor for v    local descriptor for i    parameter descriptor for x

while (i < v.length)

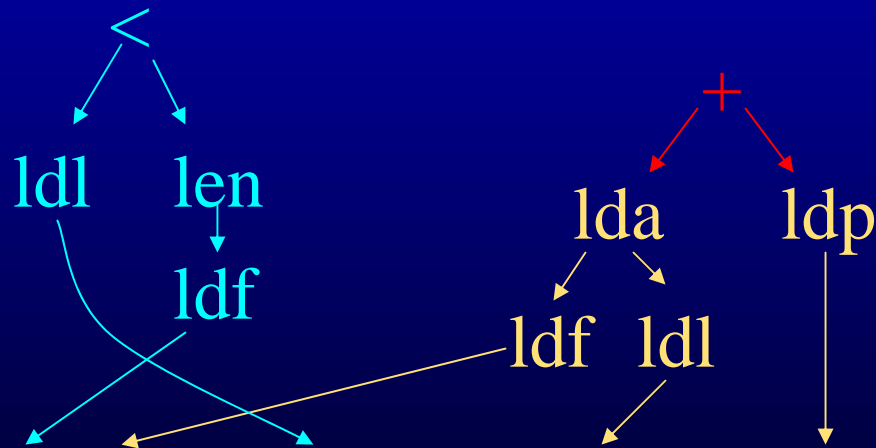
v[i] = v[i]+x;



field descriptor for v    local descriptor for i    parameter descriptor for x

while (i < v.length)

v[i] = v[i] + x;



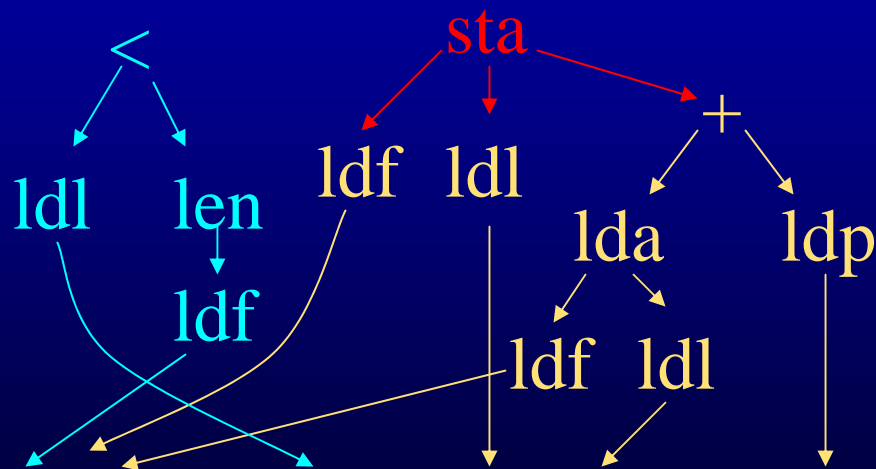
field descriptor for v   local descriptor for i   parameter descriptor for x





while (i < v.length)

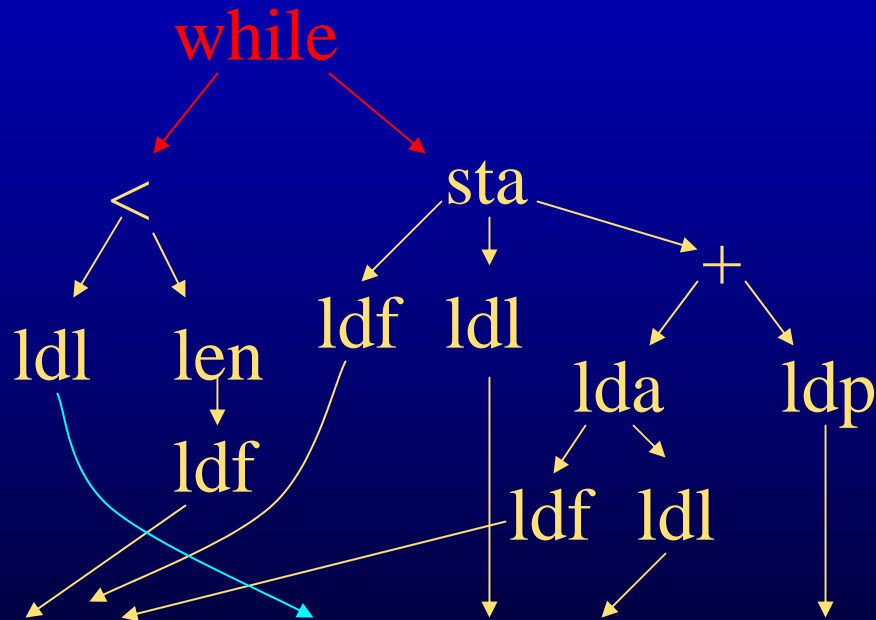
v[i] = v[i]+x;



field descriptor for v    local descriptor for i    parameter descriptor for x

**while** (i < v.length)

v[i] = v[i]+x;

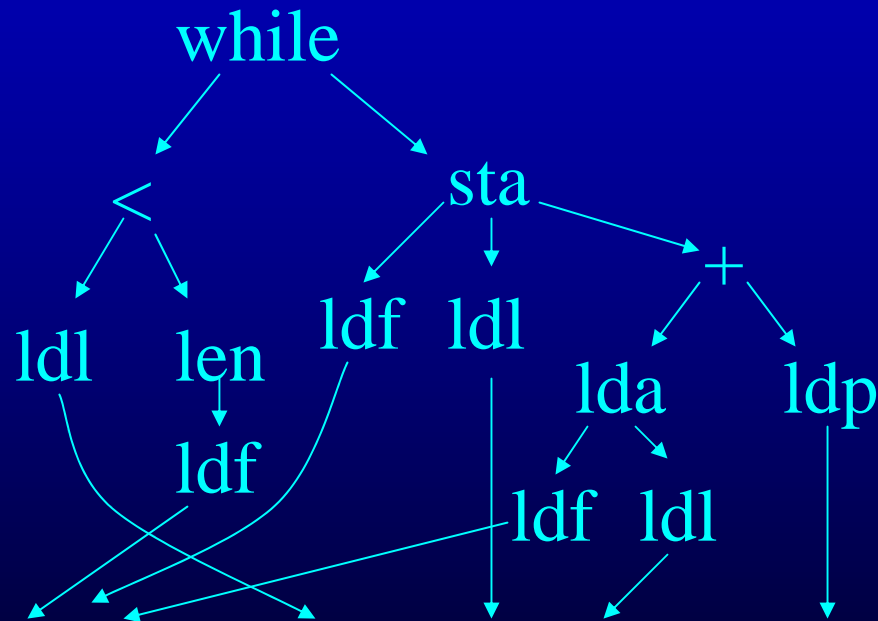


field descriptor for v    local descriptor for i    parameter descriptor for x



while (i < v.length)

v[i] = v[i]+x;

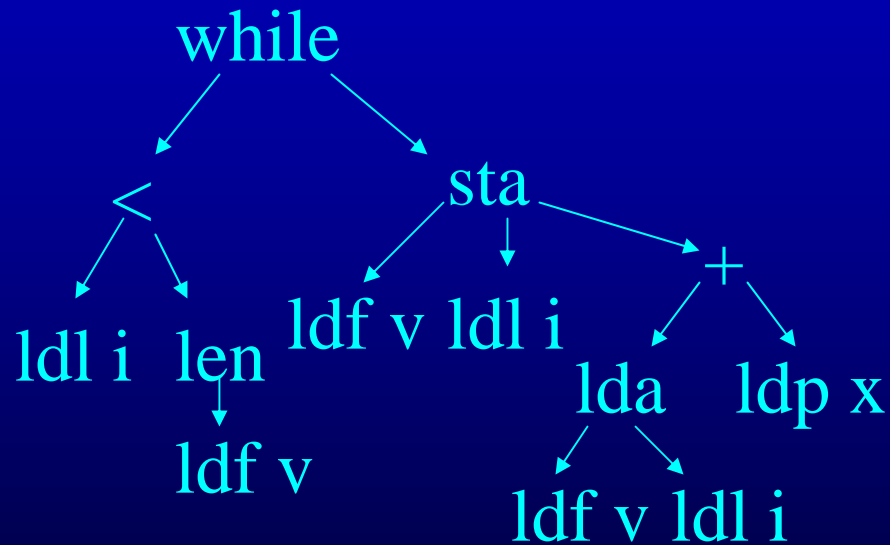


field descriptor for v    local descriptor for i    parameter descriptor for x

# Abbreviated Notation

while (i < v.length)

v[i] = v[i]+x;



# From Abstract Syntax Trees to IR

- Recursively Traverse Abstract Syntax Tree
- Build Up Representation Bottom-Up Manner
  - Look Up Variable Identifiers in Symbol Tables
  - Build Load Nodes to Access Variables
  - Build Expressions Out of Load Nodes and Operator Nodes
  - Build Store Nodes for Assignment Statements
  - Combine Store Nodes with Flow of Control Nodes

# Summary

## High-Level Intermediate Representation

- Goal: represent program in an intuitive way that supports future compilation tasks
- Representing program data
  - Symbol tables
  - Hierarchical organization
- Representing computation
  - Expression trees
  - Various types of load and store nodes
  - Structured flow of control
- Traverse abstract syntax tree to build IR

# Dynamic Dispatch

```
if (x == 0) {  
    p = new point();  
} else if (x < 0) {  
    p = new cartesianPoint();  
} else if (x > 0) {  
    p = new polarPoint();  
}  
y = p.distance();
```

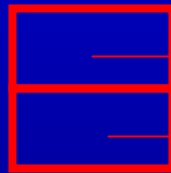
Which distance method is invoked?

- if p is a point  
return(0)
- if p is a cartesianPoint  
return(x\*x + y\*y)
- if p is a polarPoint  
return(r\*r)
- Invoked Method Depends on Type of Receiver!

# Implementing Dynamic Dispatch

- Basic Mechanism: Method Table

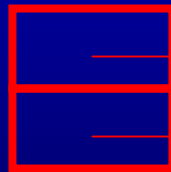
method table for  
point objects



getColor method for point

distance method for point

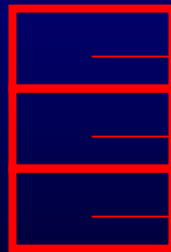
method table for  
cartesianPoint objects



getColor method for point

distance method for cartesianPoint

method table for  
polarPoint objects



getColor method for point

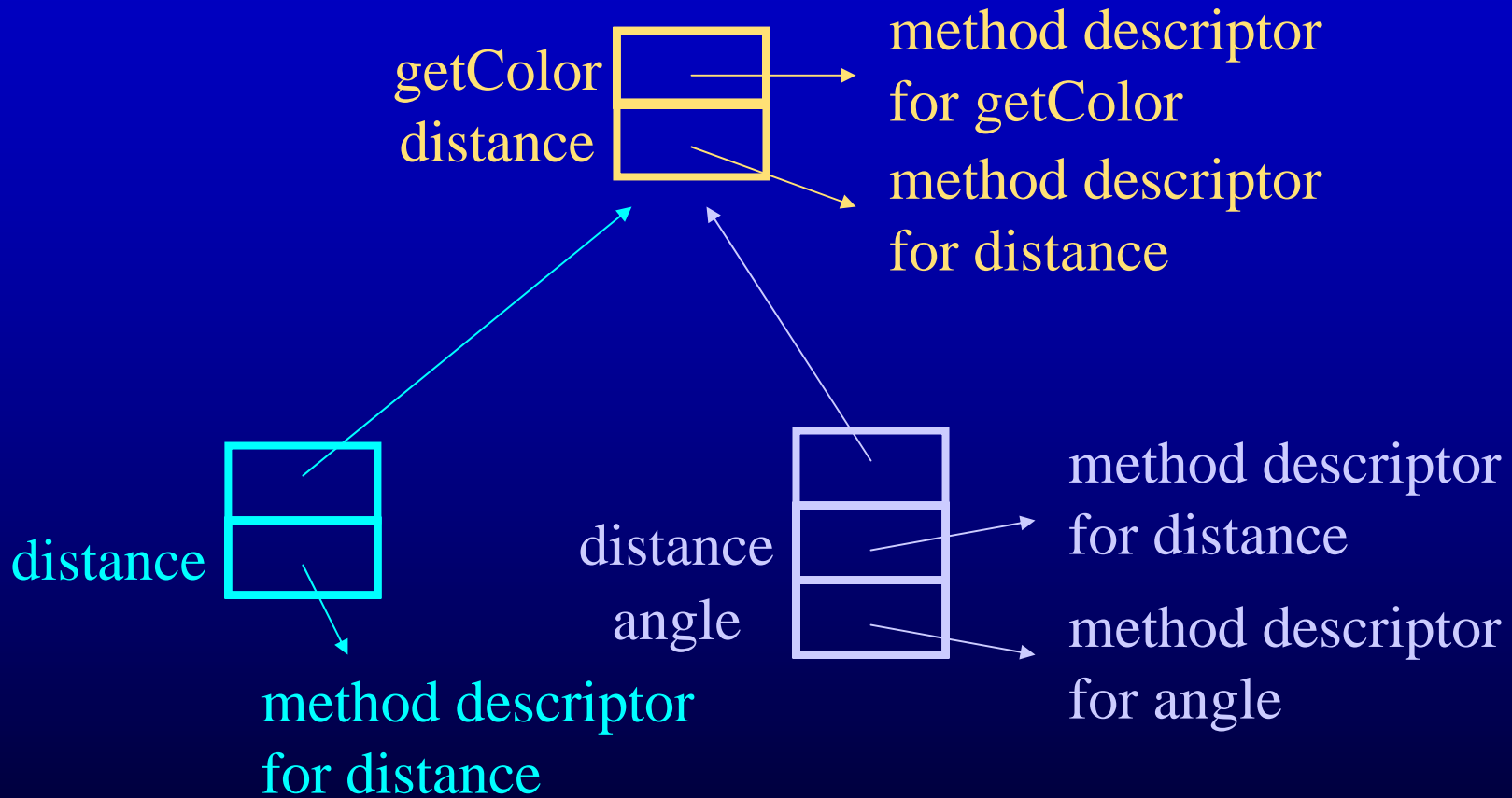
distance method for polarPoint

angle method for polarPoint

# Invoking Methods

- Compiler Numbers Methods In Each Inheritance Hierarchy
  - getColor is Method 0, distance is Method 1, angle is Method 2
- Method Invocation Sites Access Corresponding Entry in Method Table
- Works For Single Inheritance Only
  - not for multiple inheritance, multiple dispatch, or interfaces

# Hierarchy in Method Symbol Tables for Points





# Lookup In Method Symbol Tables

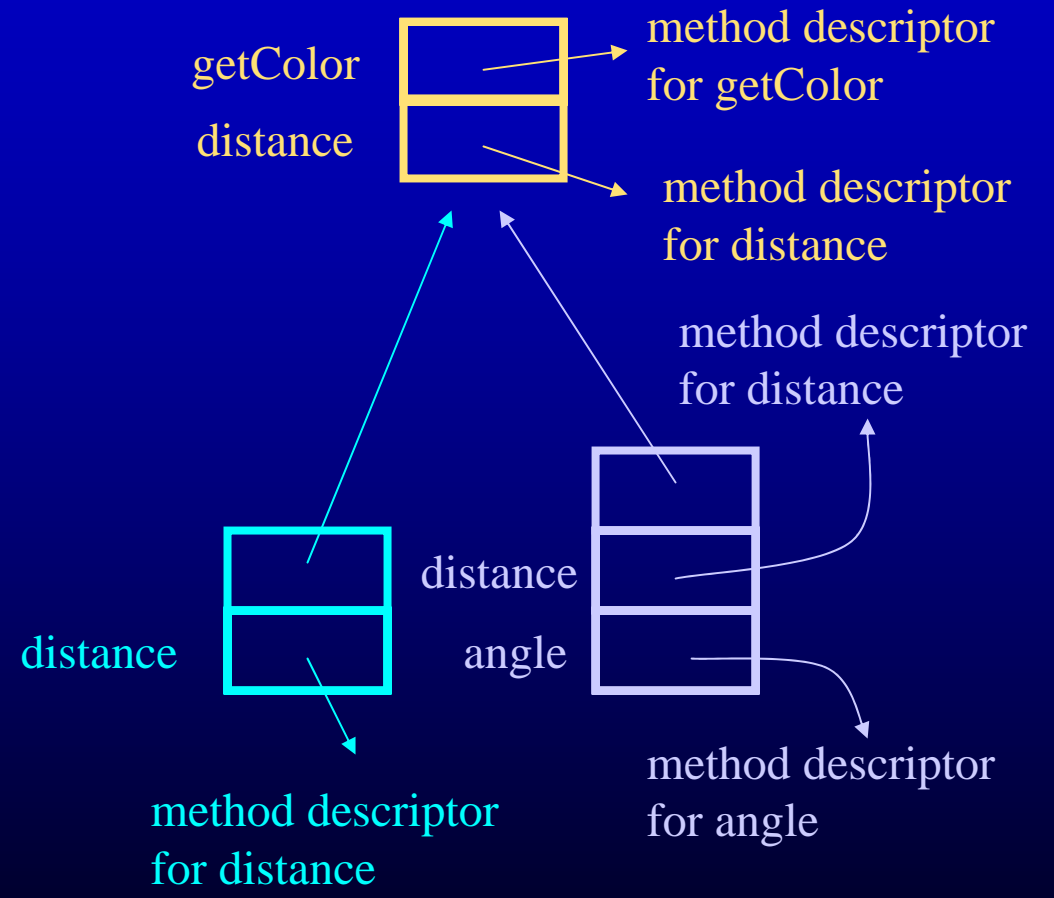
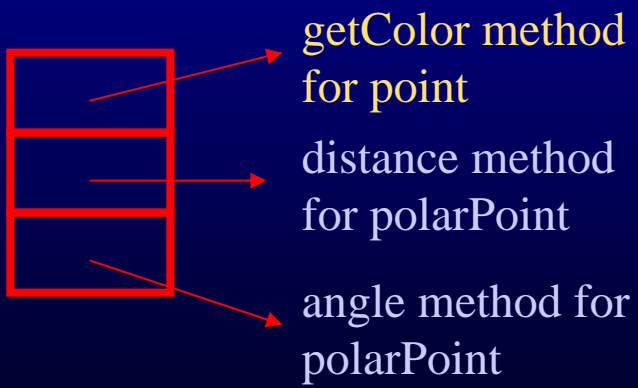
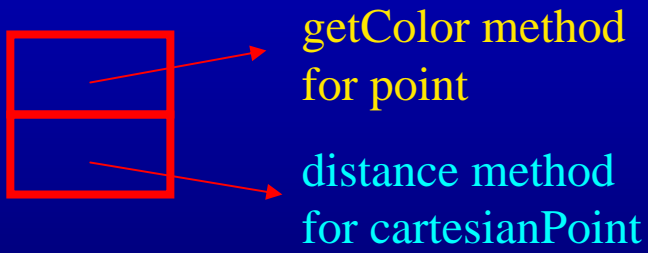
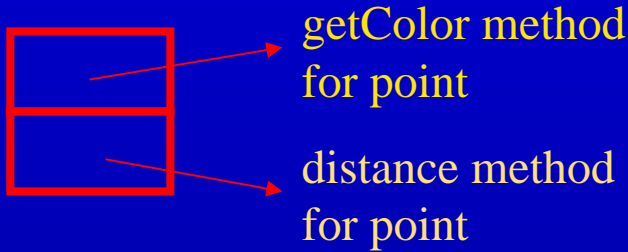
- Starts with method table of declared class of receiver object
- Goes up class hierarchy until method found
  - `point p; p = new point(); p.distance();`
    - finds distance in point method symbol table
  - `point p; p = new cartesianPoint(); p.distance();`
    - finds distance in point method symbol table
  - `cartesianPoint p; p = new cartesianPoint(); p.getColor();`
    - finds getColor in point method symbol table

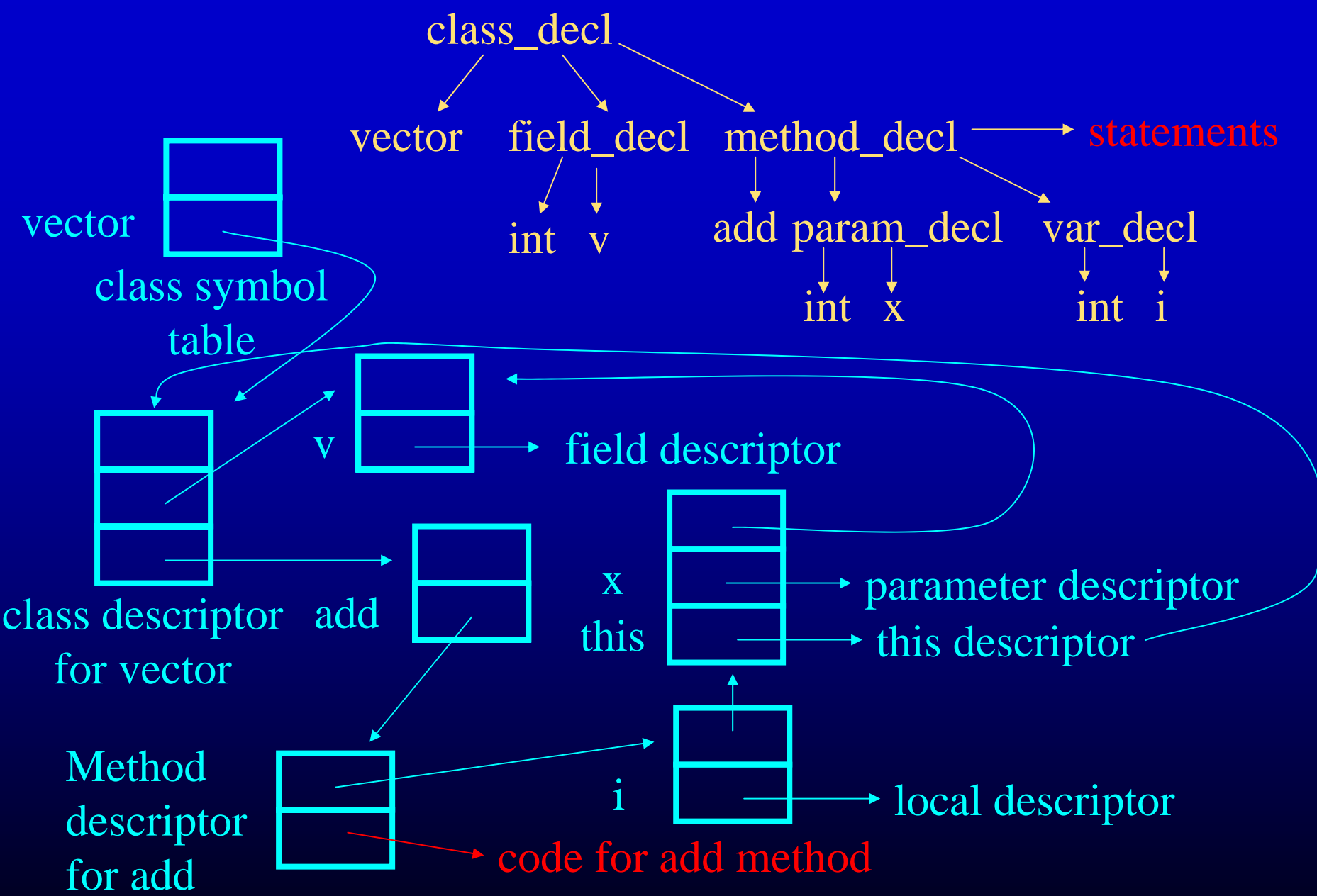
# Static Versus Dynamic Lookup

- Static lookup done at compile time for type checking and code generation
- Dynamic lookup done when program runs to dispatch method call
- Static and dynamic lookup results may differ!
  - `point p; p = new cartesianPoint(); p.distance();`
    - Static lookup finds distance in point method table
    - Dynamic lookup invokes distance in cartesianPoint class
    - Dynamic dispatch mechanism used to make this happen

# Static and Dynamic Tables

- **Static Method Symbol Table**
  - Used to look up method definitions at compile time
  - Index is method name
  - Lookup starts at method symbol table determined by declared type of receiver object
  - Lookup may traverse multiple symbol tables
- **Dynamic Method Table**
  - Used to look up method to invoke at run time
  - Index is method number
  - Lookup simply accesses a single table element

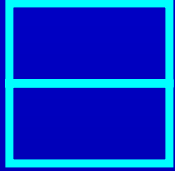




# Eliminating Parse Tree Construction

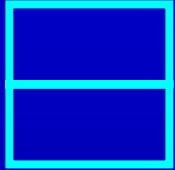
- Parser actions build symbol tables
  - Reduce actions build tables in bottom-up fashion
  - Actions correspond to activities that take place in top-down fashion in parse tree traversal
- Eliminates intermediate construction of parse tree - improves performance
- Also less code to write (but code may be harder to write than if just traverse parse tree)

```
class vector { int v[]; void add(int x) { int i; ... } }
```



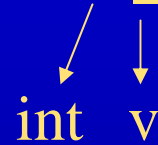
class symbol  
table

```
class vector { int v[]; void add(int x) { int i; ... } }
```



class symbol  
table

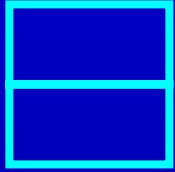
field\_decl



field descriptor

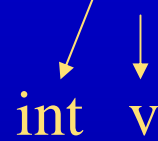


```
class vector { int v[]; void add(int x) { int i; ... } }
```



class symbol  
table

field\_decl



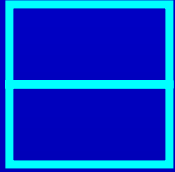
param\_decl



field descriptor

parameter descriptor

```
class vector { int v[]; void add(int x) { int i; ... } }
```



class symbol  
table

field\_decl

int v

param\_decl

int x

var\_decl

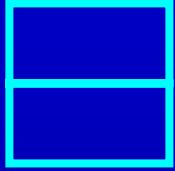
int i

field descriptor

parameter descriptor

local descriptor

```
class vector { int v[]; void add(int x) { int i; ... } }
```



class symbol  
table

field\_decl

int v

field descriptor

param\_decl

int x

parameter descriptor

statements

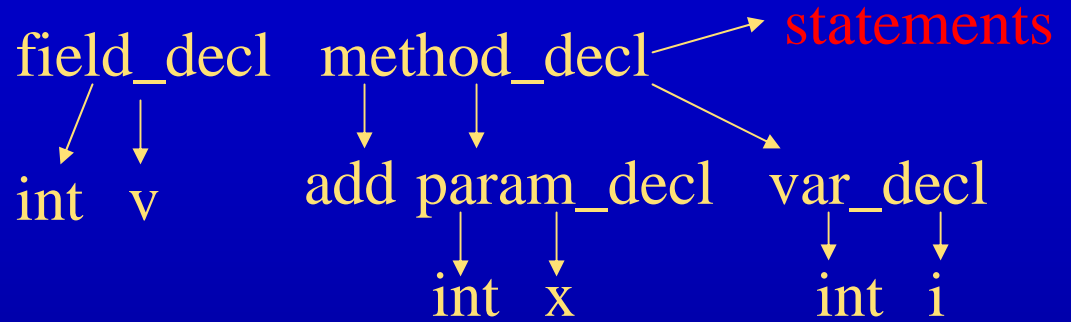
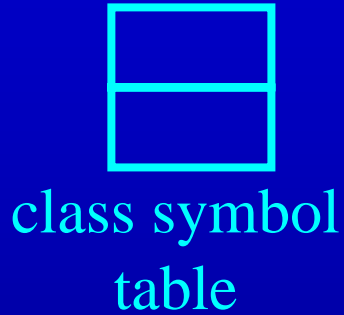
var\_decl

int i

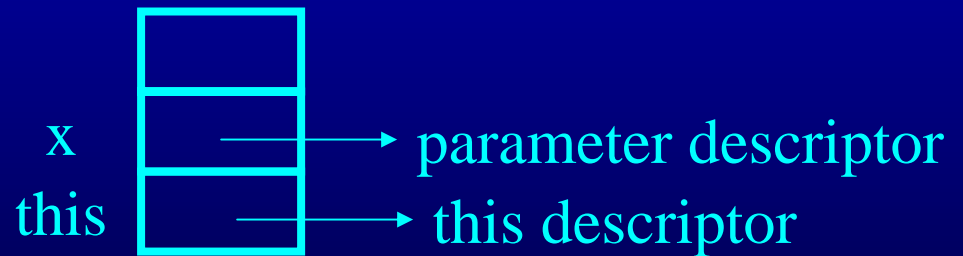
local descriptor

code for add method

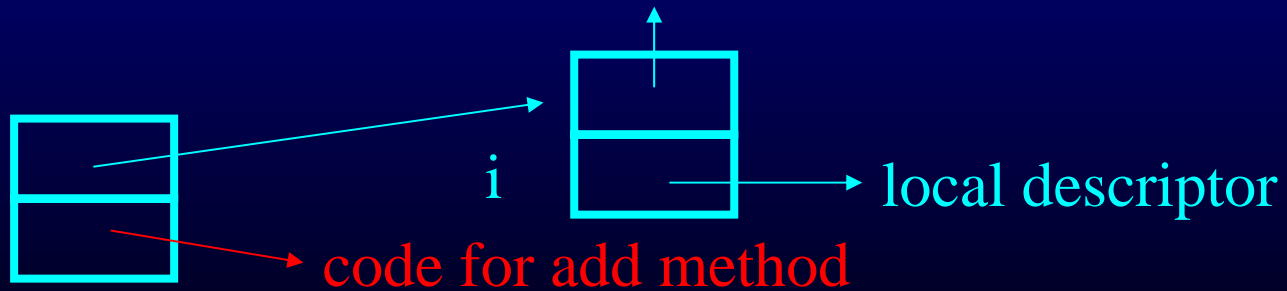
```
class vector { int v[]; void add(int x) { int i; ... } }
```



field descriptor

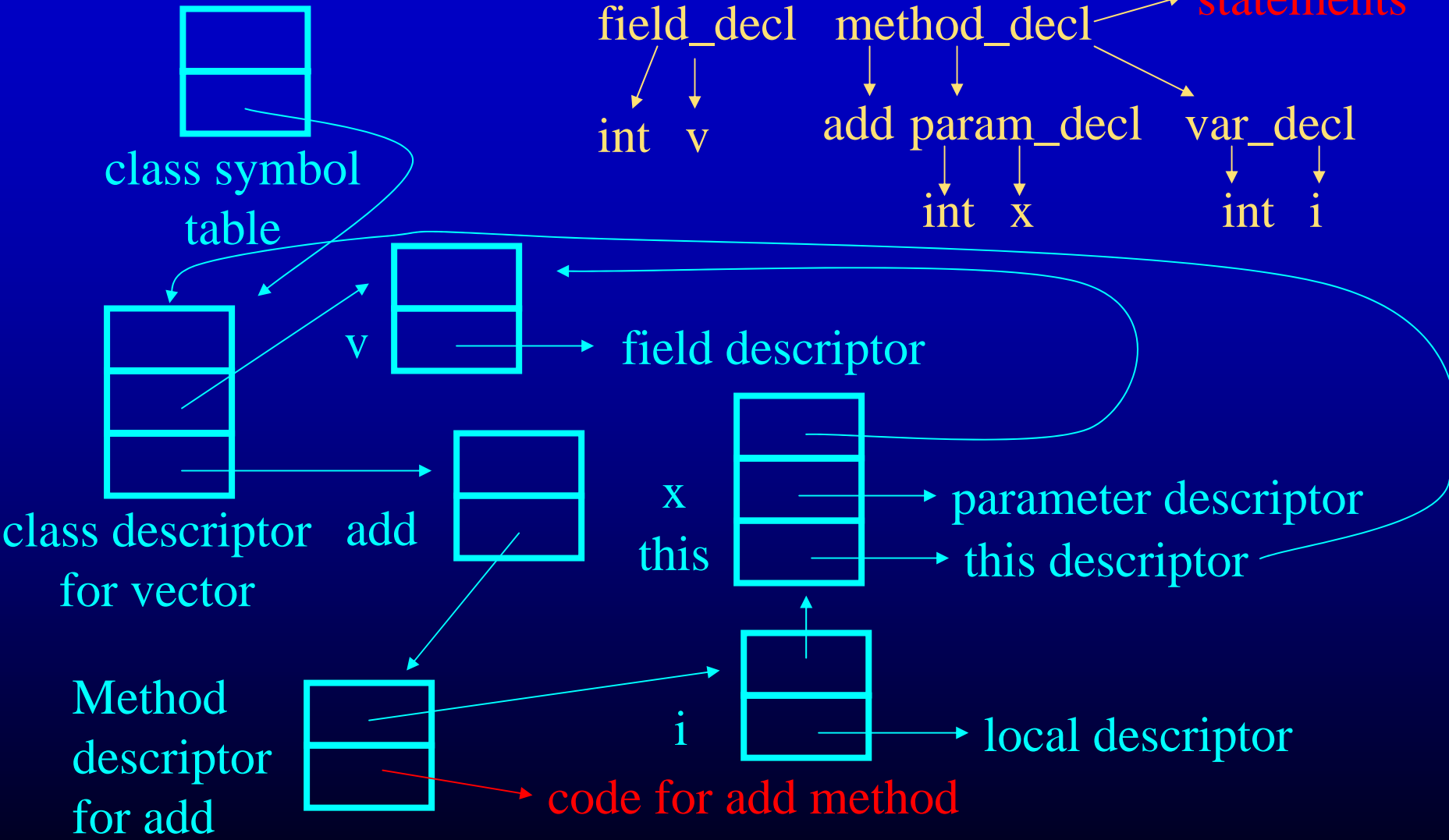
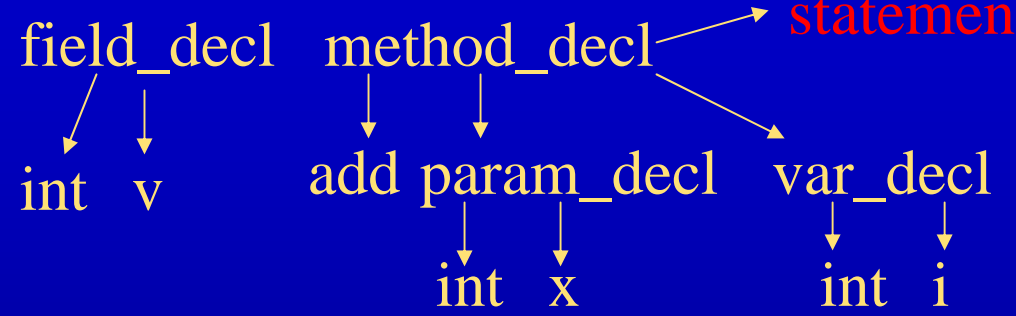


Method descriptor for add



```
class vector { int v[]; void add(int x) { int i; ... } }
```

statements



# Nested Scopes

- So far, have seen several kinds of nesting
  - Method symbol tables nested inside class symbol tables
  - Local symbol tables nesting inside method symbol tables
- Nesting disambiguates potential name clashes
  - Same name used for class field and local variable
  - Name refers to local variable inside method

# Nested Code Scopes

- Symbol tables can be nested arbitrarily deeply with code nesting:

```
class bar {  
    baz x;  
    int foo(int x) {  
        double x = 5.0;  
        { float x = 10.0;  
            { int x = 1; ... x ...}  
            ... X ...  
        }  
        ... X ...  
    }  
}
```

Note: Name clashes with nesting can reflect programming error. Compilers often generate warning messages if it occurs.

# What is a Parse Tree?

- Parse Tree Records Results of Parse
- External nodes are terminals/tokens
- Internal nodes are non-terminals

`class_decl ::= 'class' name '{' field_decl method_decl '}'`

`field_decl ::= 'int' name '[' ;`

`method_decl ::= 'void' name '(' param_decl ')'`

`'{' var_decl stats '}'`



# Abstract Versus Concrete Trees

- Remember grammar hacks
  - left factoring, ambiguity elimination, precedence of binary operators
- Hacks lead to a tree that may not reflect cleanest interpretation of program
- May be more convenient to work with abstract syntax tree (roughly, parse tree from grammar before hacks)

# Building IR Alternatives

- Build concrete parse tree in parser, translate to abstract syntax tree, translate to IR
- Build abstract syntax tree in parser, translate to IR
- Roll IR construction into parsing

# From Abstract Syntax Trees to Symbol Tables

- Recursively Traverse Tree
- Build Up Symbol Tables As Traversal Visits Nodes

# Traversing Class Declarations

- Extract Class Name and Superclass Name
- Create Class Descriptor (field and method symbol tables), Put Descriptor Into Class Symbol Table
- Put Array Descriptor Into Type Symbol Table
- Lookup Superclass Name in Class Symbol Table, Make Superclass Link in Class Descriptor Point to Retrieved Class Descriptor
- Traverse Field Declarations to Fill Up Field Symbol Table
- Traverse Method Declarations to Fill Up Method Symbol Table