Lecture 17

CIS 341: COMPILERS

Announcements

- Project 4 is due tonight!
- Project 5 Compiling objects in full Oat
  - Will be available soon
  - Due April 4th
  - Next Tuesday's lecture will include discussion of the typechecking rules for the project
- Project 6 (Optimizations)
  - Due: April 16th
- Project 7 (Oat programming)
  - "Due" April 23rd but no penalty if submitted as late as Friday, May 3rd

- Final Exam:
  - Tuesday, April 30th noon-2:00 pm
  - Moore 216

MODULARITY & ABSTRACTION

Modules as Records

- Records (or structs) bundle values together, mapping names to values.
- Modules *also* bundle values together…
  - Except that modules are computed a *load* time
  - They are (usually) 2nd class (e.g. modules cannot be passed arguments to functions). (OCaml v. 3.12 has support for first-class modules.)
- But… module interfaces look like record types:

```
module PWC = struct
  let names : string array = …
  let passwords : string array = …
  let check_password (n:string, p:string):bool = …
  let is_name (n:string):bool = …
end :
sig
  val check_password : string * string -> bool
  val is_name : string -> bool
end
```

Abstract Data Types

- Key idea: *abstract type*
  - An identifier representing an *unknown* type

- Abstract Data Type is
  - A type identifier (possibly parameterized) +
  - Declared operations on that type +
  - Concrete type definition (a representation) +
  - Concrete implementation of the operations

Interface

Implementation

- IntSet interface in OCaml:

```
module type IntSet = sig
  type intset          (* Note: no type definition *)
  val empty : intset
  val insert : int -> intset -> intset
  val has : int -> intset -> bool
end
```

IntSet example in OCaml

```
module IntSet1 : IntSet = struct
  type intset = int list
  let empty = []
  let insert i s = i::s
  let rec has = …
end
```

This signature ascription *seals* the modules with an abstract type, hiding the representation of intset.

```
module IntSet2 : IntSet = struct
  type intset = Leaf | Node of intset * int * intset
  let empty = Leaf
  let rec insert i s = …
  let rec has = …
end
```

Implementing Abstract Types

- Representation of the abstract type is hidden from code other than the implementation itself
  - CLU, Ada, Modula-3, ML

- Because external code doesn't know representation, it can't violate the abstraction boundary
  - e.g. break representation invariants

- Positive: The same interface can be reimplemented multiple ways.
- Positive: Module signatures can bundle together multiple related abstract types.
- Negative: Compiler doesn't know representation either
  - When compiling external code it must use level of indirection
  - No stack allocation of abstract types

Type Checking A Module

- Module definitions must agree with the interface in the signature
- Inside the module the concrete types are known
  - Extend the context with the definition (or substitute S_i for I_i)
- This rule also provides width subtyping

Module

$$E' = E, \mathtt{I}_1 = \mathtt{S}_1, \mathtt{I}_2 = \mathtt{S}_2, \ldots \mathtt{I}_n = \mathtt{S}_n$$

$$\frac{E' \vdash e_1 : \mathtt{T}_1 \quad E' \vdash e_2 : \mathtt{T}_2 \quad \ldots \quad E' \vdash e_m : \mathtt{T}_m \quad E' \vdash e_{m+1} : \mathtt{T}_{m+1} \ldots E' \vdash e_k : \mathtt{T}_k}{E \vdash \texttt{struct type } I_1 = S_1 \ldots \texttt{type } I_n = S_n \texttt{ let } v_1 : T_1 = e_1 \ldots \texttt{let } v_k : T_k = e_k \texttt{ end} : \texttt{sig type } I_1 \ldots \texttt{type } I_n \texttt{ val } v_1 : T_1 \ldots \texttt{val } v_m : T_m \texttt{ end}}$$

```
E ⊢ struct                          :   sig
      type I1 = S1                        type I1
       …                                   …
      type In = Sn                        type In
      let v1 : T1 = e1                    val v1 : T1
       …                                   …
      let vk : Tk = ek                    val vm : Tm
    end                                 end
```

Classes

- Fields or instance variables:
  - Values may differ from object to object (not shared)
  - Usually mutable
  - Presence inherited from the superclass
- Methods:
  - (Function) values shared among all instances of a class
  - Code inherited from the superclass
  - Immutable (usually)
  - Usually take an implicit argument that refers to the object itself (`this` or `self`)
- All components have visibility modifiers
  - public/private/protected (subclass visible)

Objects as Abstract Data Types (ADTs)

- Objects: another way of extending records to ADTs
- Source code for the class defines the concrete types and implementation
- Interface defined either implicitly (via public members) or explicitly via interface ascription

```
class IntSet1 implements IntSet {
    private List<Integer> rep;
    public IntSet1() {
       rep = new LinkedList<Integer>();}

    public IntSet1 insert(int i) {
       rep.add(new Integer(i));
       return this;}

    public boolean has(int i) {
       return rep.contains(new Integer(i));}

    public int size() { return rep.size(); }
}
```

```
interface IntSet {
    public IntSet insert(int i);
    public boolean has(int i);
    public int size();
}
```

Classes in C++/Java

- Classes have private/public visibility qualifiers that hide part of the object.
- A class is a *partially* abstract type
  - (Note: do not confuse with Java's 'abstract' keyword)
- Interface file declares the representation
  - Method code is (mostly) hidden from the outside
- Positive: This mechanism allows external code to know how much space each object takes while still providing encapsulation
  - Objects can be stack allocated (good for cache coherence/performance)
- Negative: Change to representation can require complete recompilation, even of external code
  - C++ is notoriously slow to compile
- Negative: Each class defines only a *single* type.

IntSet example in C

- intset.h:

```
struct intset;
extern struct intset *empty;
struct intset *insert(int i, struct intset *s);
int has(int i, struct intset *s);
```

- intset.c:

```
#include "intset.h"

struct intset {struct intset *left;
       int val; struct intset *right; };

struct intset *empty = NULL;

struct intset *insert(int i, struct intset *s) {…}
int has(int i, struct intset *s) {…}
```

No Abstraction in C

- C provides hiding/encapsulation but no abstraction.

- (Unchecked) Casts allow any client code to violate the representation invariants of the module.

COMPILING CLASSES AND OBJECTS

Code Generation for Objects

- Methods:
  - Generating method body code is similar to functions/closures
  - Generating method calls requires *dispatch*
- Fields:
  - Issues are the same as for records
  - Memory layout
  - Packing & alignment
  - Generating access code
- Dynamic Types:
  - Checked downcasts
  - "instanceof" and similar type dispatch

Multiple Implementations

- The same interface can be implemented by multiple classes:

```
interface IntSet {
    public IntSet insert(int i);
    public boolean has(int i);
    public int size();
}
```

```
class IntSet1 implements IntSet {
  private List<Integer> rep;
  public IntSet1() {
    rep = new LinkedList<Integer>();}

  public IntSet1 insert(int i) {
    rep.add(new Integer(i));
    return this;}

  public boolean has(int i) {
    return rep.contains(new Integer(i));}

  public int size() {return rep.size();}
}
```

```
class IntSet2 implements IntSet {
  private Tree rep;
  private int size;
  public IntSet2() {
    rep = new Leaf(); size = 0;}

  public IntSet2 insert(int i) {
    Tree nrep = rep.insert(i);
    if (nrep != rep) {
      rep = nrep; size += 1;
    }
    return this;}

  public boolean has(int i) {
    return rep.find(i);}

  public int size() {return size;}
}
```

The Dispatch Problem

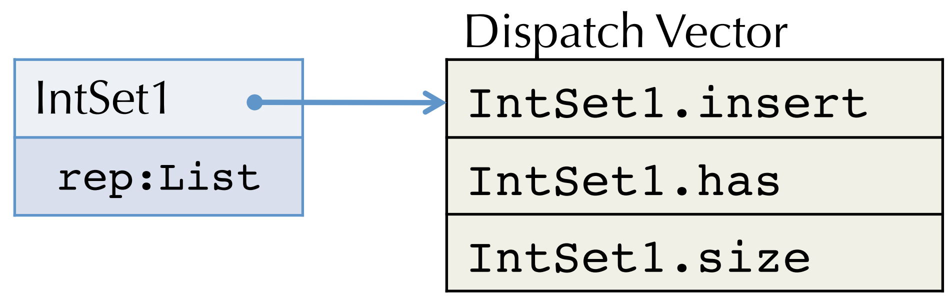
- Consider a client program that uses the IntSet interface:

```
IntSet set = …;
int x = set.size();
```

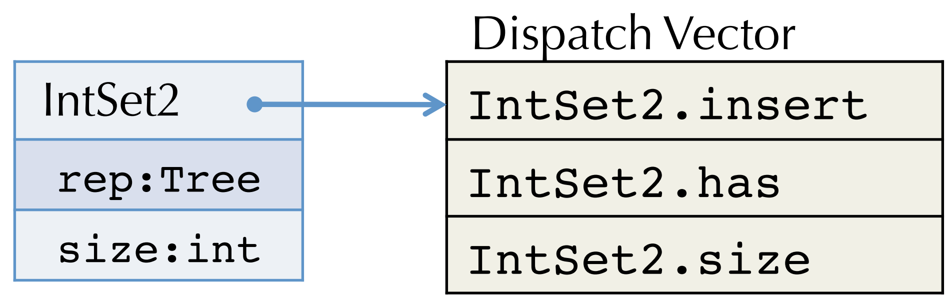
- Which code to call?
  - `IntSet1.size` ?
  - `IntSet2.size` ?
- Client code doesn't know the answer.
  - So objects must "know" which code to call.
  - Invocation of a method must indirect through the object.

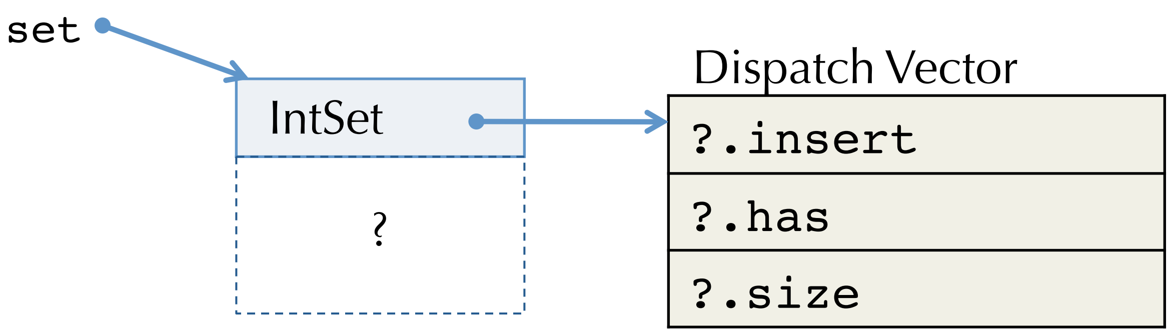
Compiling Objects

- Objects contain a pointer to a *dispatch vector* (also called a *virtual table* or *vtable*) with pointers to method code.



- Code receiving `set:IntSet` only knows that `set` has an initial dispatch vector pointer and the layout of that vector.





Method Dispatch (Single Inheritance)

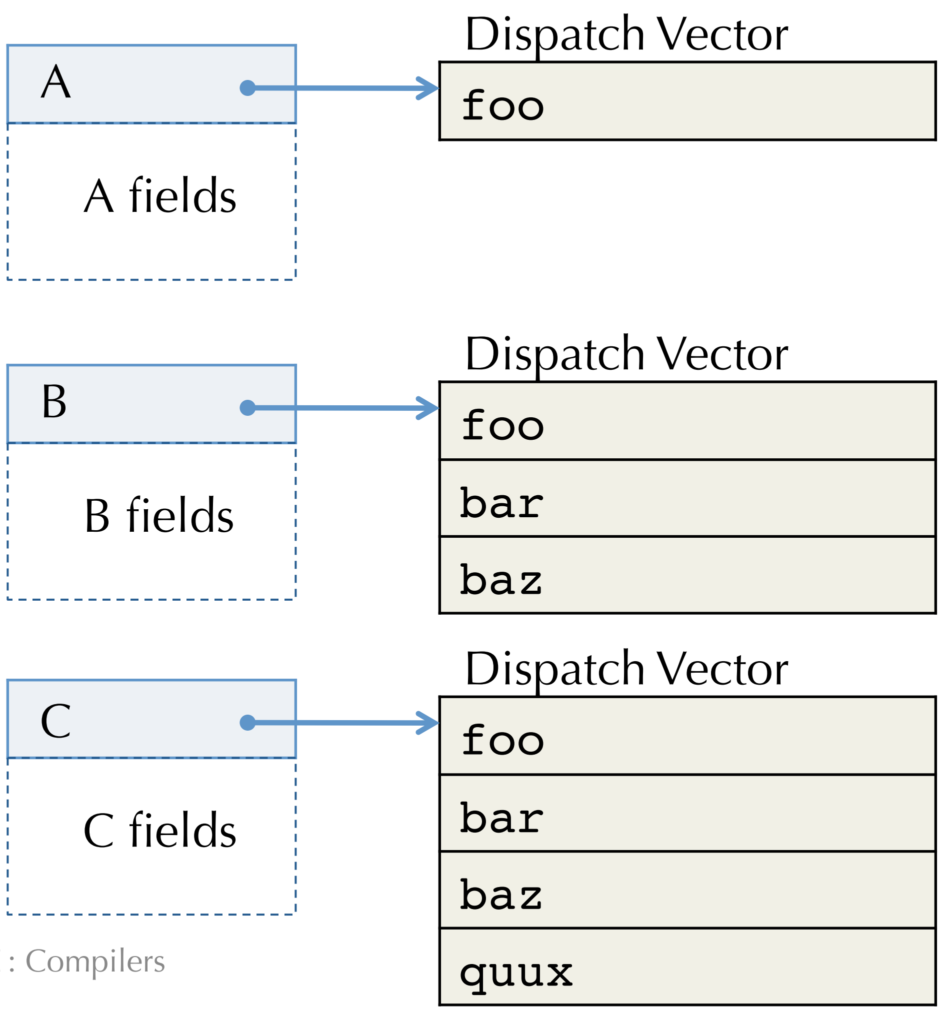
- Idea: every method has its own small integer index.
- Index is used to look up the method in the dispatch vector.

	Index
`interface A {` `  void foo();` `}`	0
`interface B extends A {` `  void bar(int x);` `  void baz();` `}`	1 2
`class C implements B {` `  void foo() {…}` `  void bar(int x) {…}` `  void baz() {…}` `  void quux() {…}` `}`	0 1 2 3

Inheritance / Subtyping:
A <: B <: C

Dispatch Vector Layouts

- Each interface and class gives rise to a dispatch vector layout.
- Note that inherited methods have identical dispatch indices in the subclass.



Method Arguments

- Methods bodies are compiled just like top-level procedures…
- … except that they have an implicit extra argument: `this` or `self`
  - Historically (Smalltalk), these were called the "receiver object"
  - Method calls were thought of a sending "messages" to "receivers"

A method in a class...

```
class IntSet1 implements IntSet {
   …
  IntSet1 insert(int i) { <body> }
}
```

… is compiled like this (top-level) procedure:

```
IntSet1 insert(IntSet1 this, int i) { <body> }
```

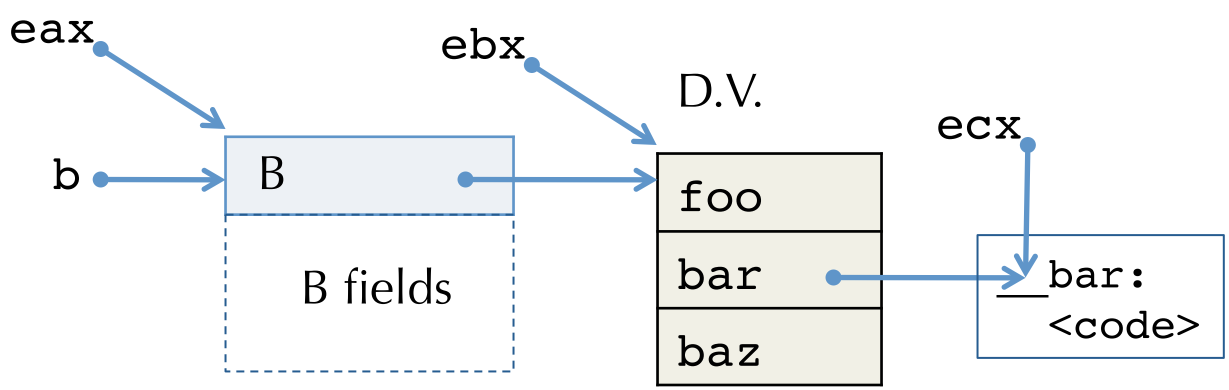
- Note 1: the type of "`this`" is the class containing the method.
- Note 2: references to fields inside <body> are compiled like `this.field`

Method Invocation Compilation

- Consider method invocation: $C \vdash [\![\texttt{e.f(e}_1\texttt{,…,e}_n\texttt{)}]\!]$
- First, compile $C \vdash [\![\texttt{e}]\!]$ to get a (reference to) an object value.
  - Call this value obj
- Push the method arguments on the stack (right-to-left).
- Push the `this` argument (it's just obj) on to the stack.
- Compute dispatch vector address into a temporary
  - dv = [obj] (just dereference obj)
- Execute: Call [dv + 4*i]
  - Where i is method f's dispatch vector index i

X86 Code For Dynamic Dispatch

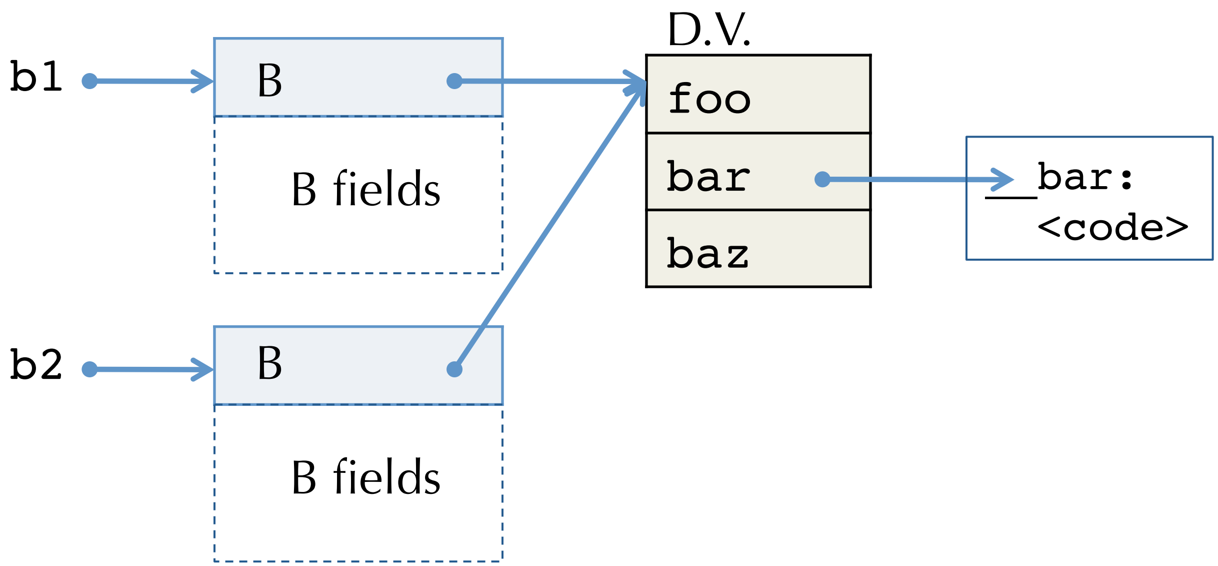
- Suppose `b : B`
- What code for `b.bar(3)`?
  - `bar` has index 1
  - Offset = 4 * 1




```
Mov eax, ⟦b⟧
Push 3              // Method argument
Push eax            // "this" pointer
Mov ebx, [eax]
Mov ecx, [ebx + 4]  // D.V. + offset
Call ecx
```

Sharing Dispatch Vectors

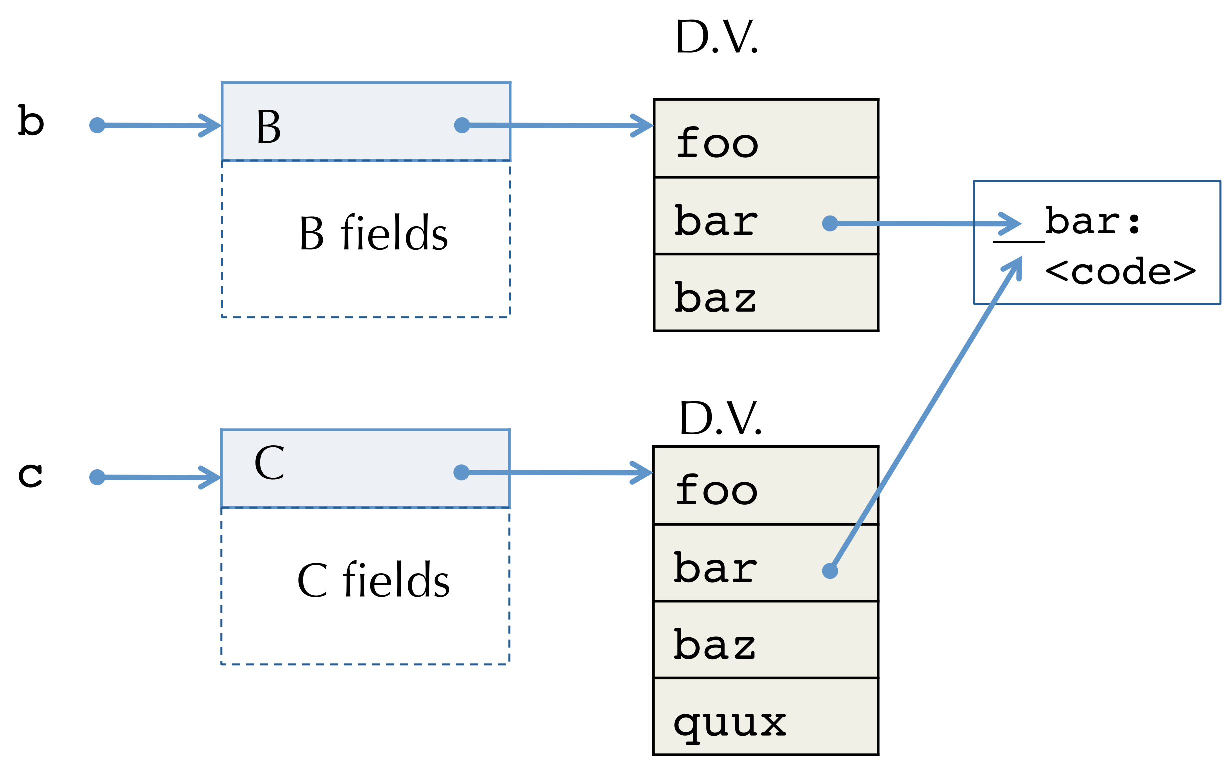
- All instances of a class may share the same dispatch vector.
  - Assuming that methods are immutable.
- Code pointers stored in the dispatch vector are available at link time – dispatch vectors can be built once at link time.



- One job of the object constructor is to fill in the object's pointer to the appropriate dispatch vector.
- Note: The address of the D.V. *is* the run-time representation of the object's type.

Inheritance: Sharing Code

- Inheritance: Method code "copied down" from the superclass
  - If not overridden in the subclass
- Works with separate compilation – superclass code not needed.



MULTIPLE INHERITANCE

Multiple Inheritance

- C++: a class may declare more than one superclass.
- Semantic problem: Ambiguity

```
class A { int m(); }
class B { int m(); }
class C extends A,B {…}     // which m?
```

  - Same problem can happen with fields.
  - In C++, fields and methods can be duplicated when such ambiguity arises (though explicit sharing can be declared too)

- Java: a class may implement more than one interface.
  - No semantic ambiguity: if two interfaces contain the same method declaration, then the class will implement a single method

```
interface A { int m(); }
interface B { int m(); }
class C implements A,B {int m() {…}}     // only one m
```

Dispatch Vector Layout Strategy Breaks

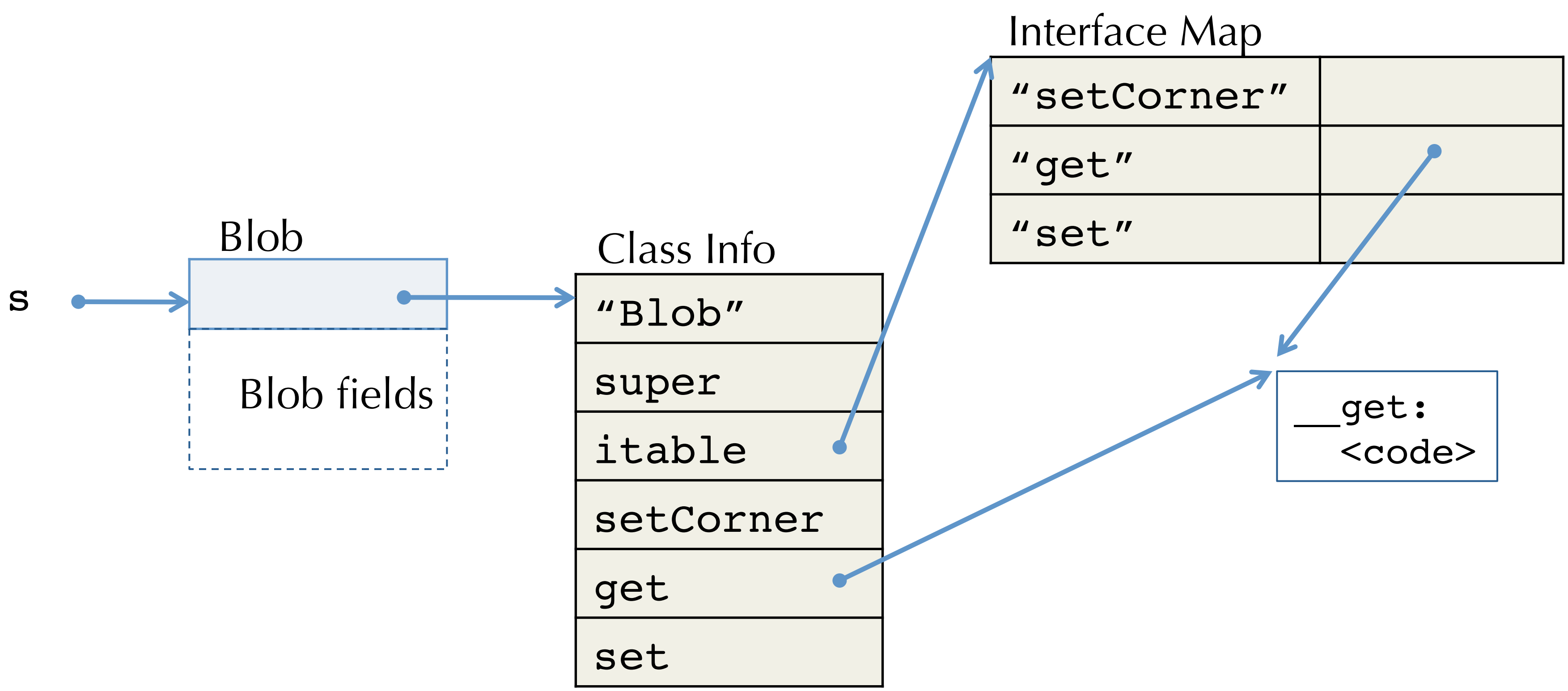
```
interface Shape {                             D.V.Index
  void setCorner(int w, Point p);                 0
}

interface Color {
  float get(int rgb);                             0
  void set(int rgb, float value);                 1
}

class Blob implements Shape, Color {
  void setCorner(int w, Point p) {…}              0?
  float get(int rgb) {…}                          0?
  void set(int rgb, float value) {…}              1?
}
```

General Approaches

- Can't directly identify methods by position anymore.
- Option 1: Use a level of indirection:
  - Map method identifiers to code pointers (e.g. index by method name)
  - Use a hash table
  - May need to do search up the class hierarchy
- Option 2: Give up separate compilation
  - Use "sparse" dispatch vectors, or binary decision trees
  - Must know then entire class hierarchy
- Option 3: Allow multiple D.V. tables (C++)
  - Choose which D.V. to use based on static type
  - Casting from/to a class may require run-time operations
- Note: many variations on these themes
  - Different Java compilers pick different approaches…

Option 1: Search + Inline Cache

- For each class & interface keep a table mapping method names to method code
  - Recursively walk up the hierarchy looking for the method name
- Note: Identifiers are in quotes are not strings; in practice they are some kind of unique identifier.



Inline Cache Code

- Optimization: At call site, store class and code pointer in a cache
  - On method call, check whether class matches cached value
- Compiling: `Shape s = new Blob();   s.get();`

  Call site 434

Table in data seg.

```
cacheClass434:
  "Blob"
cacheCode434:
  <ptr>
```

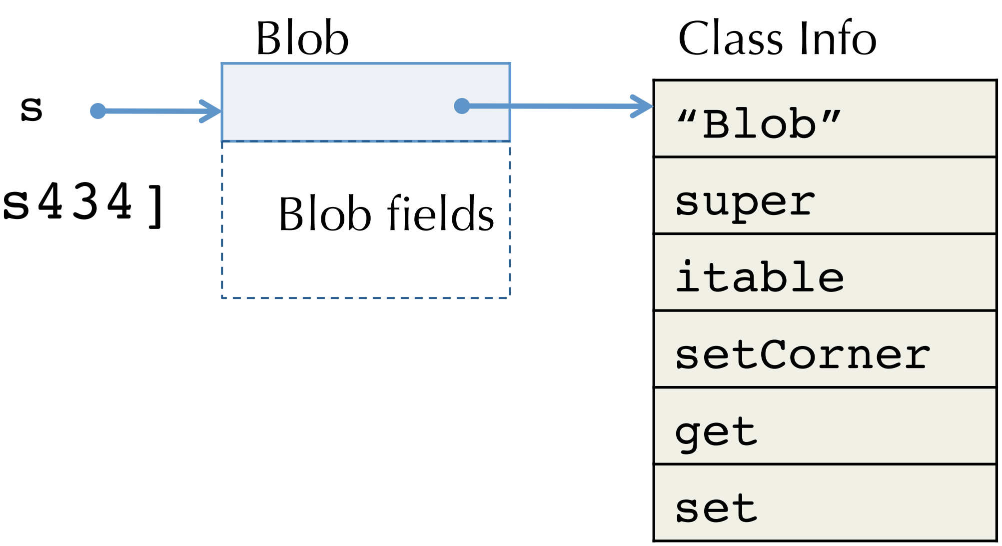
- Compiler knows that s is a Shape
  - Suppose `EAX` holds object pointer

- Cached interface dispatch:

// push parameters

```
   Mov tmp, [EAX]
   Cmp tmp, [cacheClass434]
   Jnz __miss434
   Call [cacheCode434]
__miss434:
```

// do the slow search



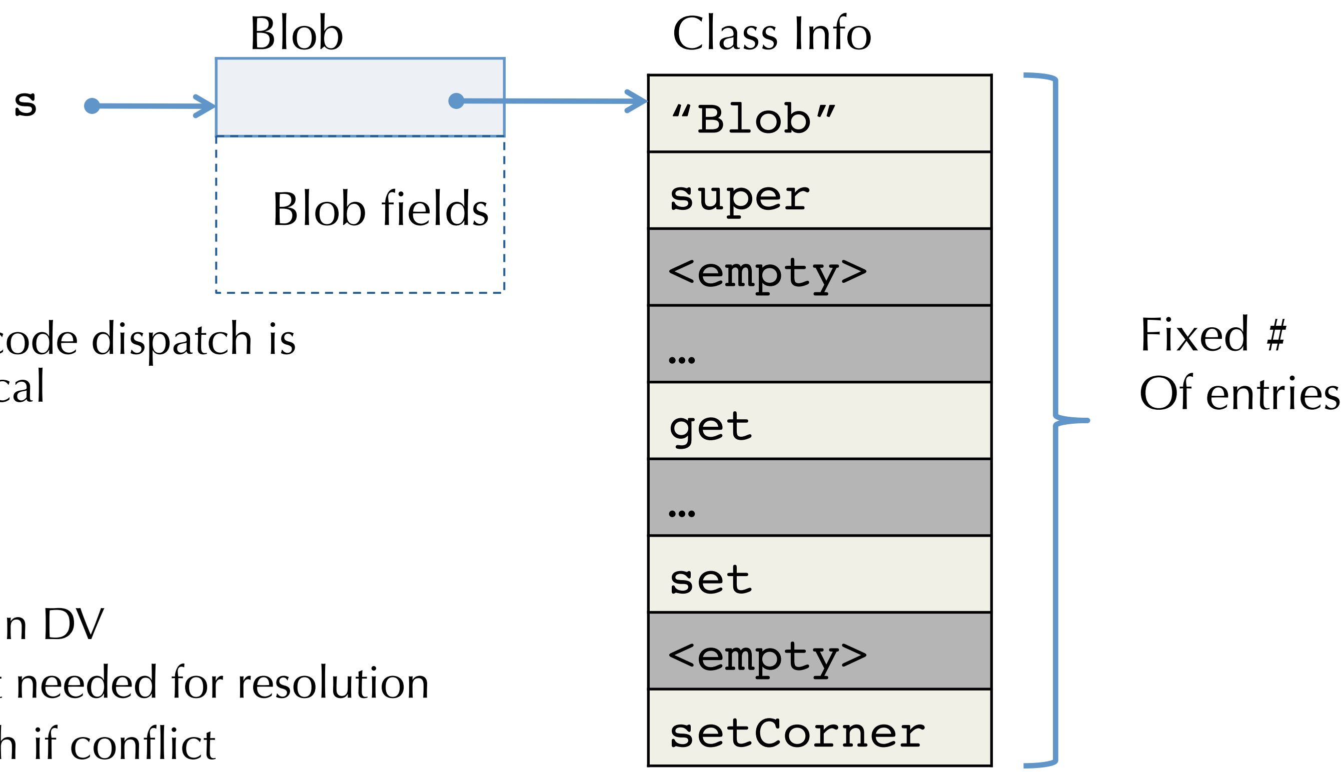
Option 1 variant 2: Hash Table

- Idea: don't try to give all methods unique indices
  - Resolve conflicts by checking that the entry is correct at dispatch
- Use hashing to generate indices
  - Range of the hash values should be relatively small
  - Hash indices can be pre computed, but passed as an extra parameter

```
interface Shape {                          D.V.Index
  void setCorner(int w, Point p);          hash("setCorner") = 11
}

interface Color {
  float get(int rgb);                      hash("get") = 4
  void set(int rgb, float value);          hash("set") = 7
}

class Blob implements Shape, Color {
  void setCorner(int w, Point p) {…}            11
  float get(int rgb) {…}                        4
  void set(int rgb, float value) {…}            7
}
```

Dispatch with Hash Tables

- What if there is a conflict?
  - Entries containing several methods point to code that resolves conflict (e.g. by searching through a table based on class name)



- Advantage:
  - Simple, basic code dispatch is (almost) identical
  - Reasonably efficient
- Disadvantage:
  - Wasted space in DV
  - Extra argument needed for resolution
  - Slower dispatch if conflict

Option 2 variant 1: Sparse D.V. Tables

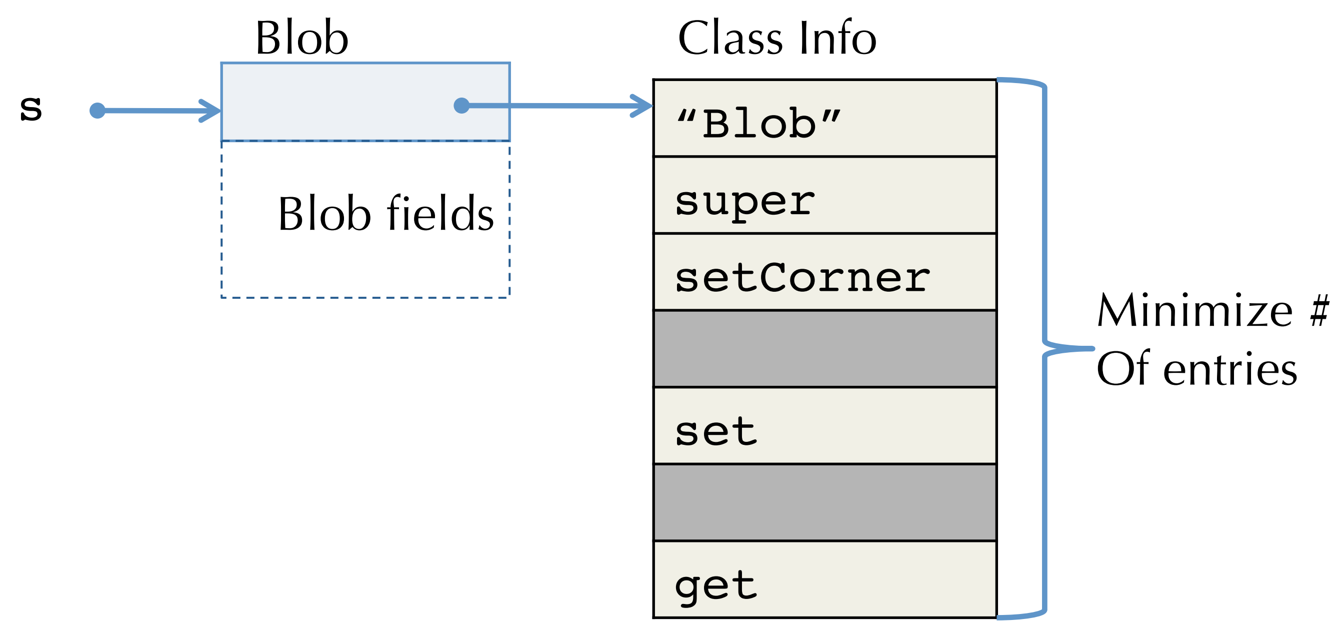
- Give up on separate compilation…
- Now we have access to the whole class hierarchy.

- So: ensure that no two methods in the same class are allocated the same D.V. offset.
  - Allow holes in the D.V. just like the hash table solution
  - Unlike hash table, there is never a conflict!

- Compiler needs to construct the method indices
  - Graph coloring techniques can be used to construct the D.V. layouts in a reasonably efficient way (to minimize size)
  - Finding an optimal solution is NP complete!

Example Object Layout

- Advantage: Identical dispatch and performance to single-inheritance case
- Disadvantage: Must know entire class hierarchy

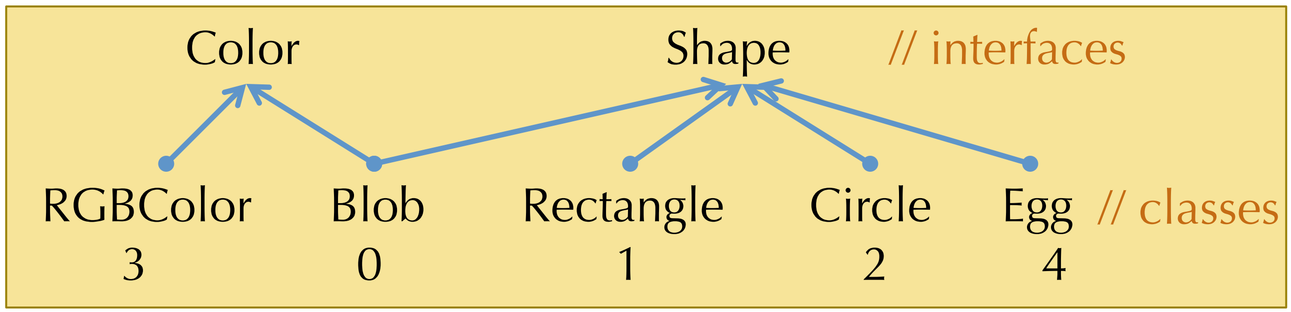


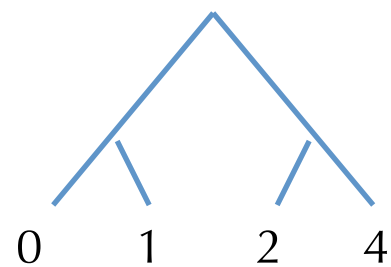
Option 2 variant 2: Binary Search Trees

- Idea: Use conditional branches not indirect jumps
- Each object has a class index (unique per class) as first word
  - Instead of D.V. pointer (no need for one!)
- Method invocation uses range tests to select among *n* possible classes in *lg n* time
  - Direct branches to code at the leaves.

```
Shape x;
x.SetCorner(…);
```

```
  Mov eax, ⟦x⟧
  Mov ebx, [eax]
  Cmp ebx, 1
  Jle   __L1
  Cmp ebx, 2
  Je __CircleSetCorner
  Jmp __EggSetCorner
__L1:
  Cmp ebx, 0
  Je __BlobSetCorner
  Jmp __RectangleSetCorner
```





Decision tree

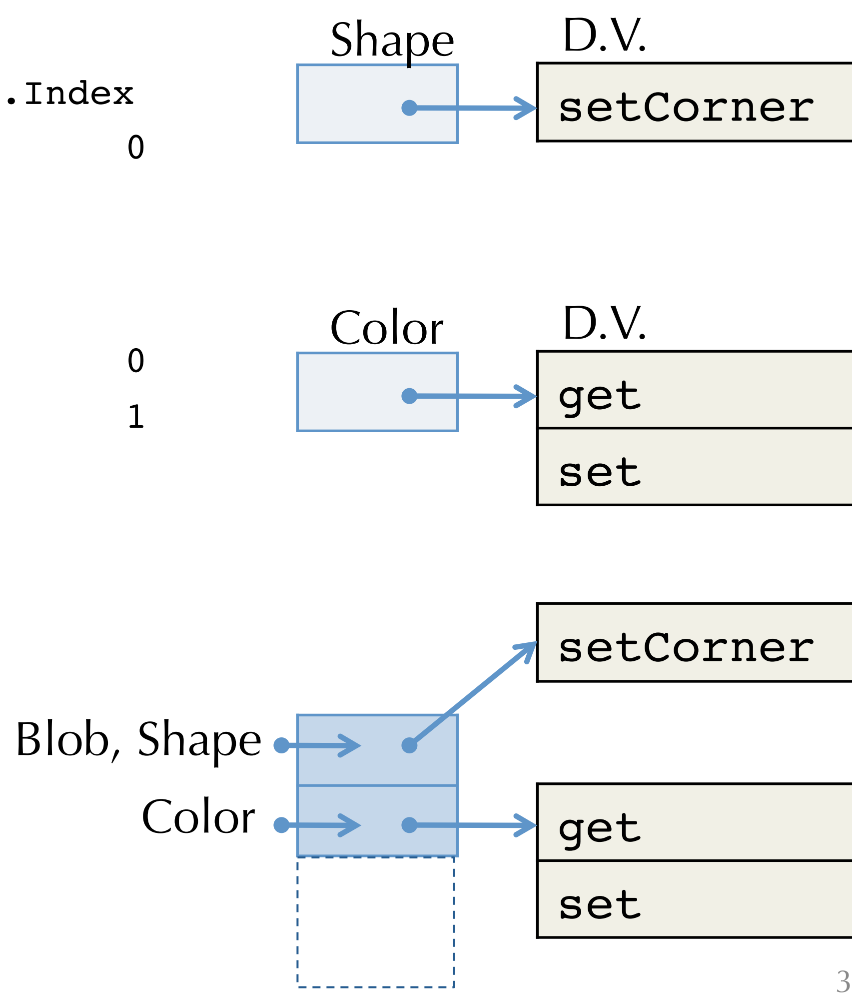
Search Tree Tradeoffs

- Binary decision trees work well if the distribution of classes that may appear at a call site is skewed.
  - Branch prediction hardware eliminates the branch stall of ~10 cycles (on X86)
- Can use profiling to find the common paths for each call site individually
  - Put the common case at the top of the decision tree (so less search)
  - 90%/10% rule of thumb: 90% of the invocations at a call site go to the same class
- Drawbacks:
  - Like sparse D.V.'s you need the whole class hierarchy to know how many leaves you need in the search tree.
  - Indirect jumps can have better performance if there are >2 classes (at most one mispredict)

Option 3: Multiple Dispatch Vectors

- Duplicate the D.V. pointers in the object representation.
- Static type of the object determines which D.V. is used.

```
interface Shape {                        D.V.Index
  void setCorner(int w, Point p);             0
}

interface Color {
  float get(int rgb);                         0
  void set(int rgb, float value);             1
}

class Blob implements Shape, Color {
  void setCorner(int w, Point p) {…}
  float get(int rgb) {…}
  void set(int rgb, float value) {…}
}
```



Multiple Dispatch Vectors

- A reference to an object might have multiple “entry points”
  - Each entry point corresponds to a dispatch vector
  - Which one is used depends on the statically known type of the program.

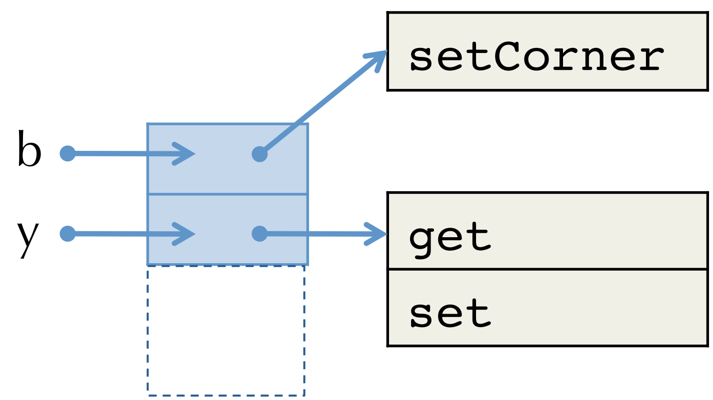
```
Blob b = new Blob();
Color y = b;      // implicit cast!
```

- Compile

  `Color y = b;`

  As

  `Mov y, ⟦b⟧ + 4`



Multiple D.V. Summary

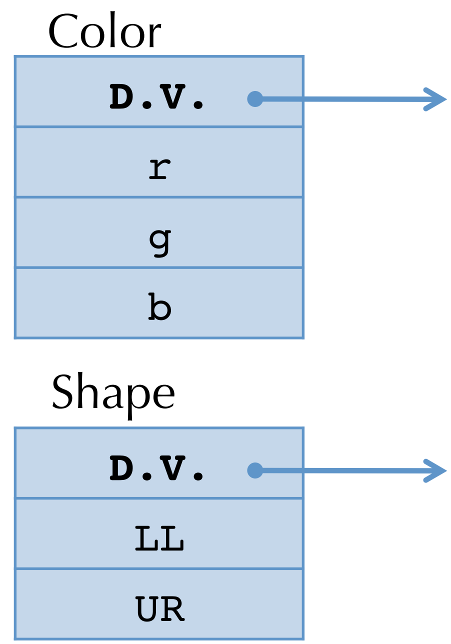
- Benefit: Efficient dispatch, same cost as for multiple inheritance
- Drawbacks:
  - Cast has a runtime cost
  - More complicated programming model… hard to understand/debug?

- What about multiple inheritance and fields?

Multiple Inheritance: Fields

- Multiple supertypes (Java): methods conflict (as we saw)
- Multiple inheritance (C++): fields can also conflict
- Location of the object's fields can no longer be a constant offset from the start of the object.

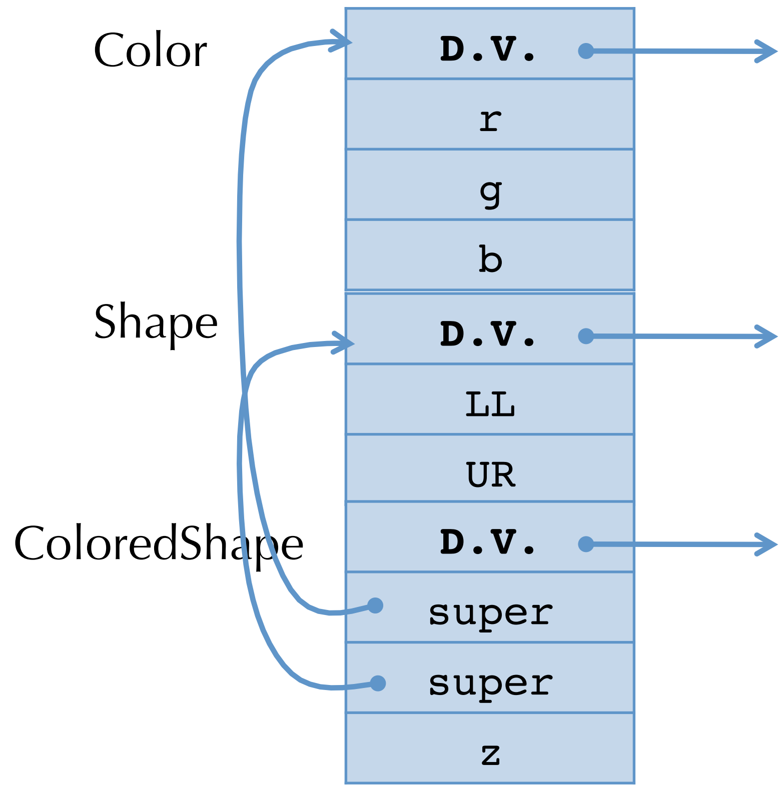
```
class Color {
  float r, g, b; /* offsets: 4,8,12 */
}
class Shape {
  Point LL, UR; /* offsets: 4, 8 */
}
class ColoredShape extends
Color, Shape {
  int z;
}
```



ColoredShape ??

C++ approach:

- Add pointers to the superclass fields
  - Need to have multiple dispatch vectors anyway (to deal with methods)
- Extra indirection needed to access superclass fields
- Used even if there is a single superclass
  - Uniformity



Compiling Static Methods

- Java supports *static* methods
  - Methods that belong to a class, not the instances of the class.
  - They have no "this" parameter (no receiver object)
- Compiled exactly like normal top-level procedures
  - No slots needed in the dispatch vectors
  - No implicit "this" parameter
- They're not really methods
  - They can only access static fields of the class

Compiling Constructors

- Java, C++ classes can declare constructors that create new objects.
  - Initialization code may have parameters supplied to the constructor
  - e.g. `new Color(r,g,b);`
- Modula-3: object constructors take no parameters
  - e.g. `new Color;`
  - Initialization would typically be done in a separate method.
- Constructors are compiled just like static methods, except:
  - The "this" variable is initialized to a newly allocated block of memory big enough to hold D.V. pointer + fields according to object layout
  - The D.V. pointer is initialized
  - The return value of the constructor is the (newly created) "this" pointer.

Observe: Closure ≈ Single-method Object

- Free variables ≈ Fields
- Environment pointer ≈ "this" parameter
- Closure for function: ≈ Instance of this class:

```
fun (x,y) ->
  x + y + a + b
```

```
class C {
  int a, b;
  int apply(x,y) {
    x + y + a + b
  }
}
```