Lecture 18

CIS 341: COMPILERS

Announcements

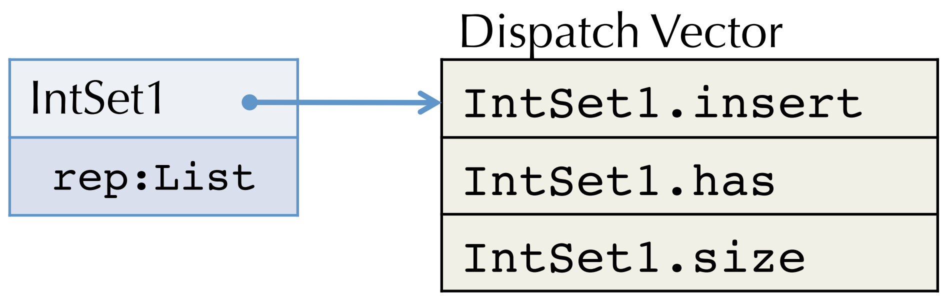
- Project 5 Compiling objects in full Oat
  - Available from the course web pages
  - Due April 8th

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

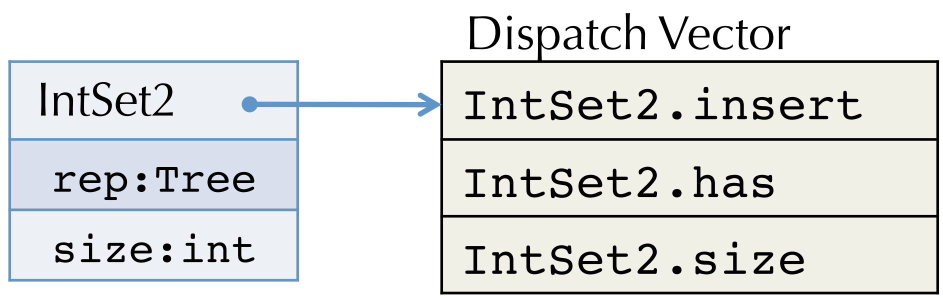
MULTIPLE INHERITANCE

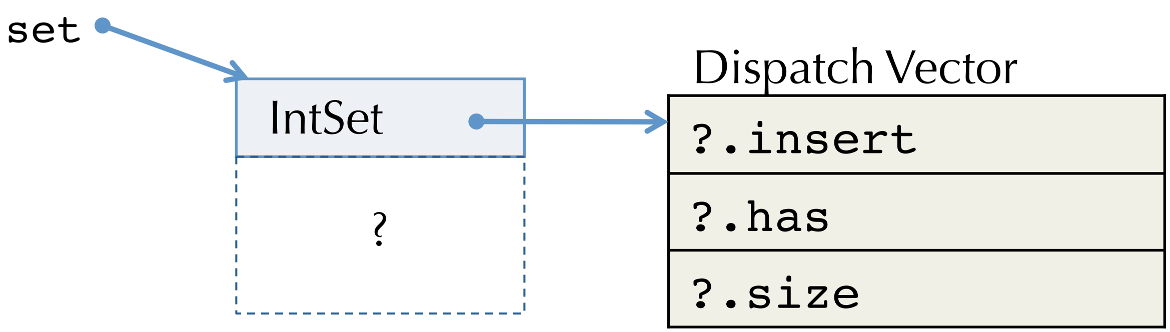
Compiling Objects

- 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

Inheritance / Subtyping:
A <: B <: C

```
class C implements B {
  void foo() {…}
  void bar(int x) {…}
  void baz() {…}
  void quux() {…}
}
```

0
1
2
3

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

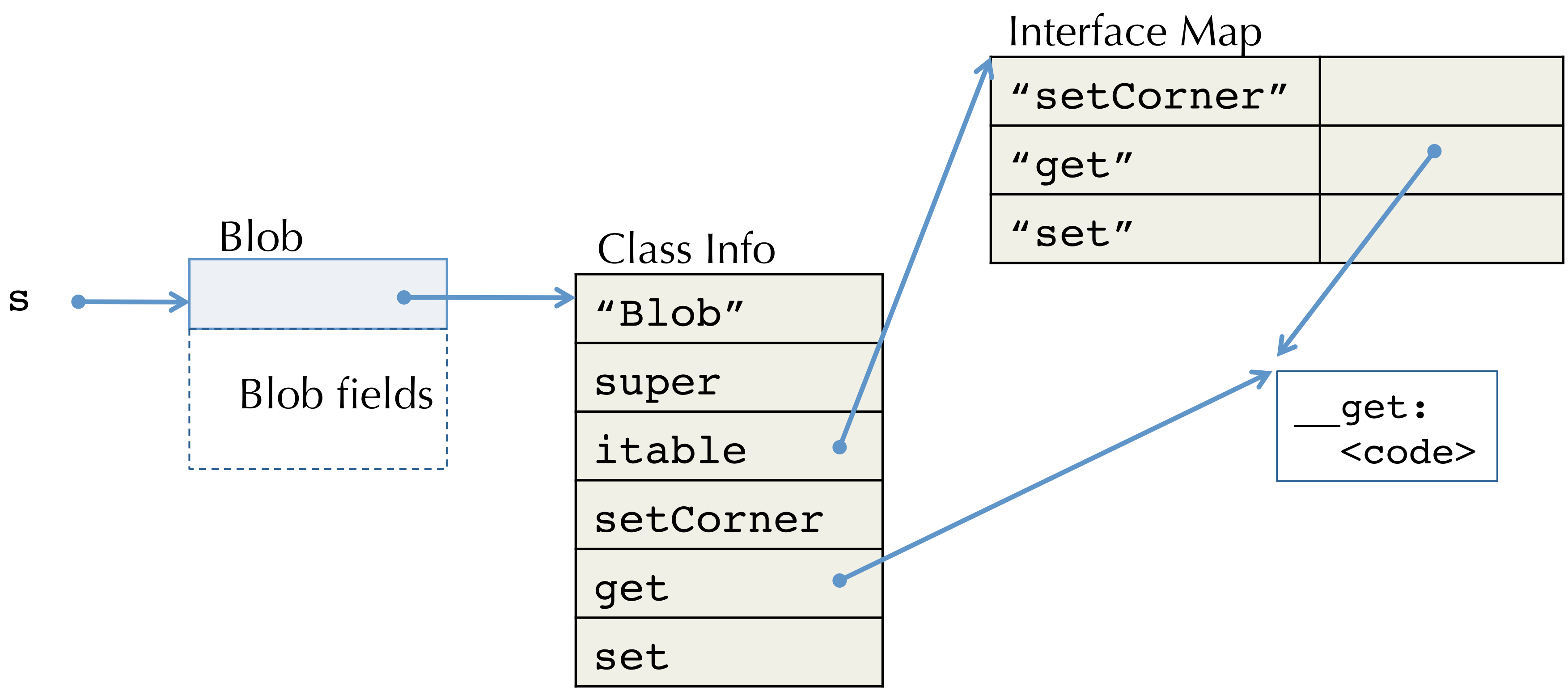
- Option 2: Give up separate compilation
  - Use "sparse" dispatch vectors, or binary decision trees
  - Must know then entire class hierarchy

- 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.

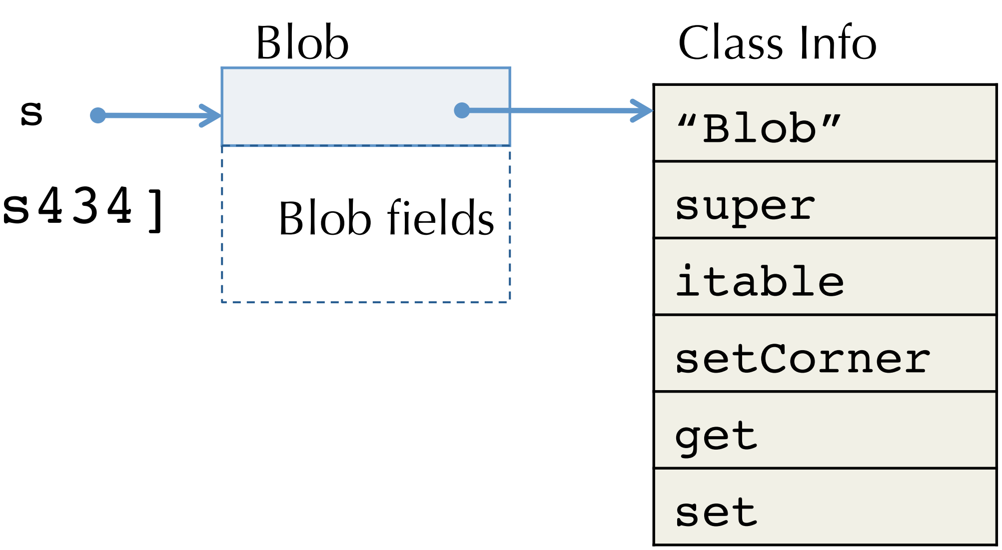
```
cacheClass434:
  “Blob”
cacheCode434:
  <ptr>
```

- 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

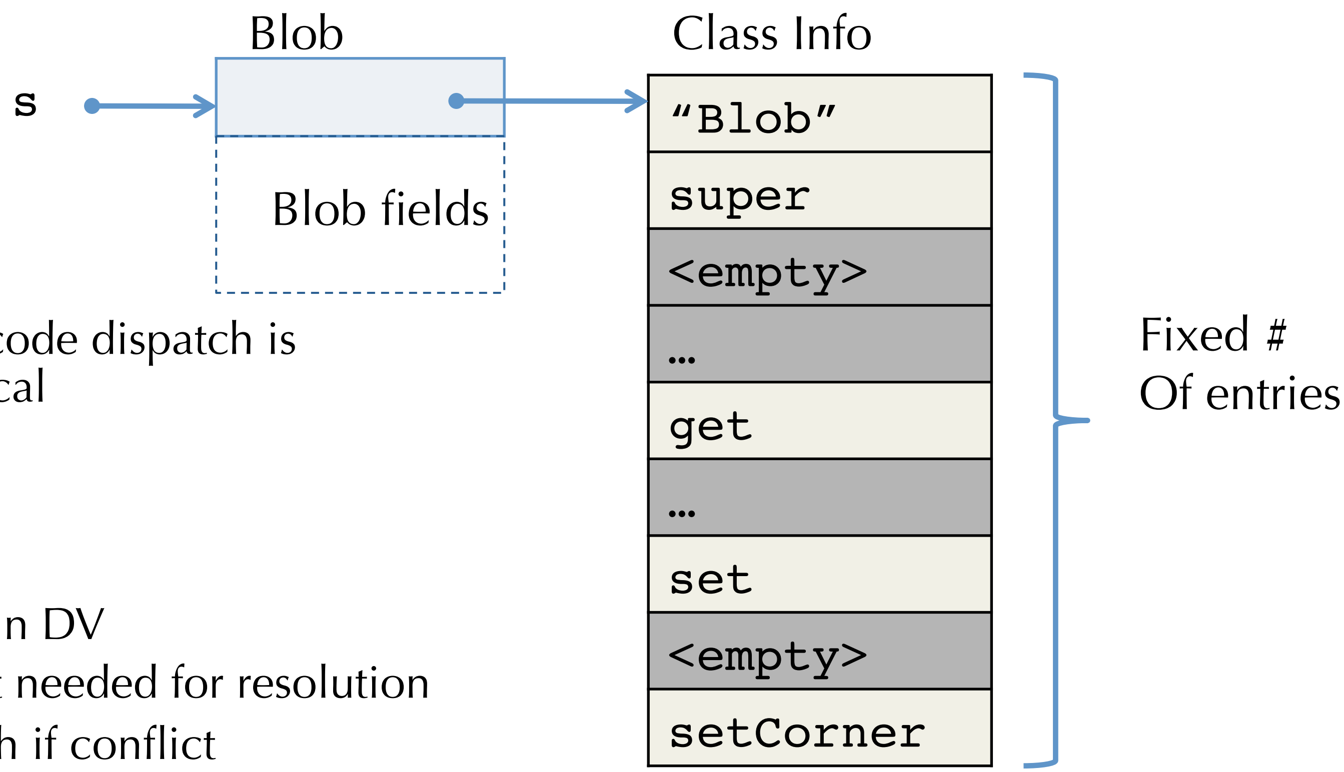
```
interface Shape {                           D.V.Index
  void setCorner(int w, Point p);           hash("setCorner") = 11
}

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

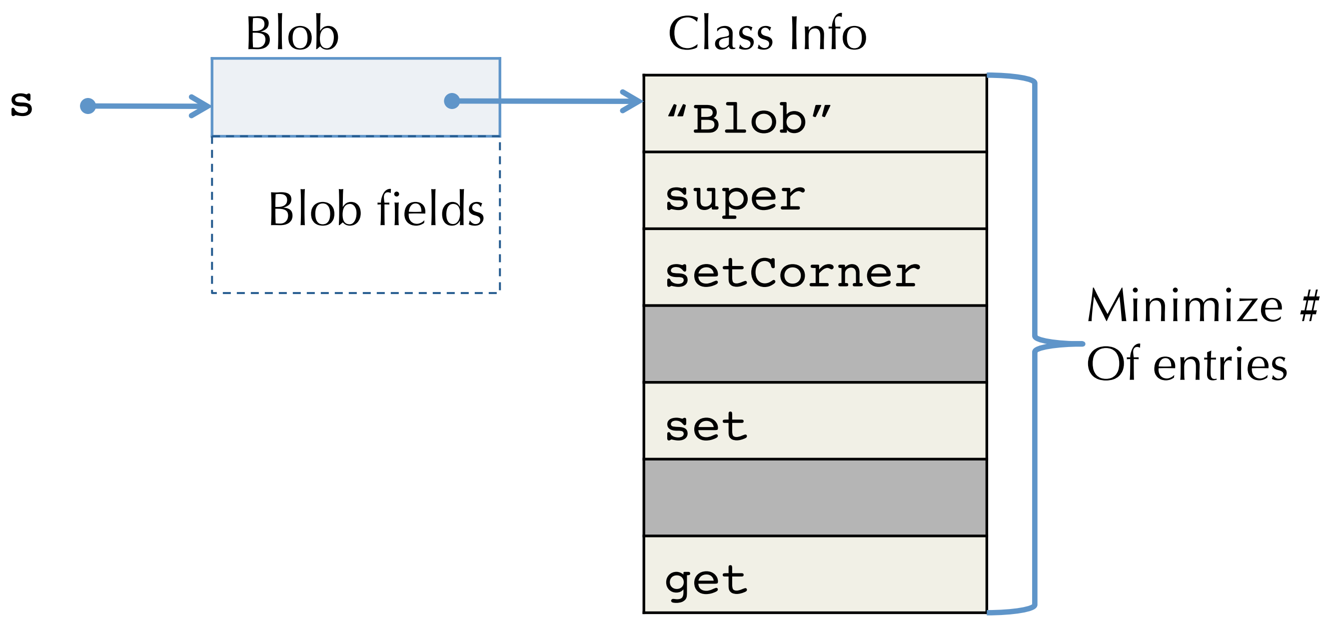
- Give up on separate compilation…
- Now we have access to the whole class hierarchy.

- 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

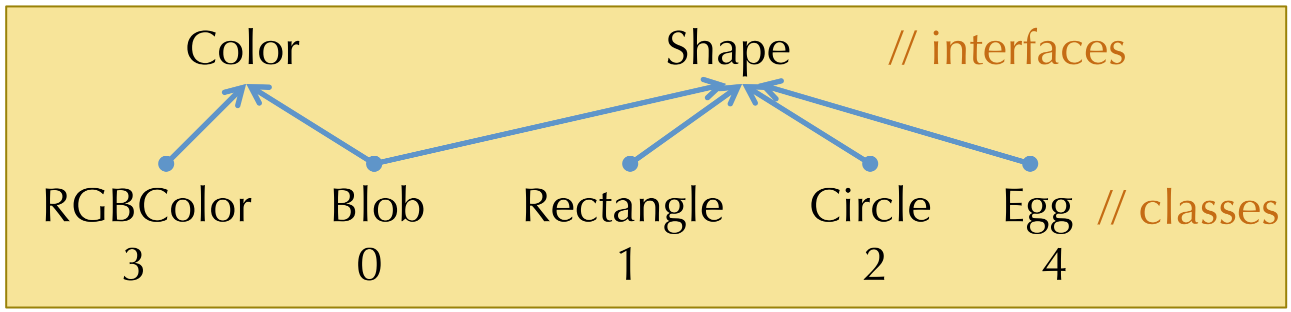


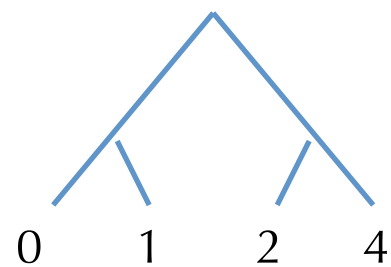
Option 2 variant 2: Binary Search Trees

- 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)

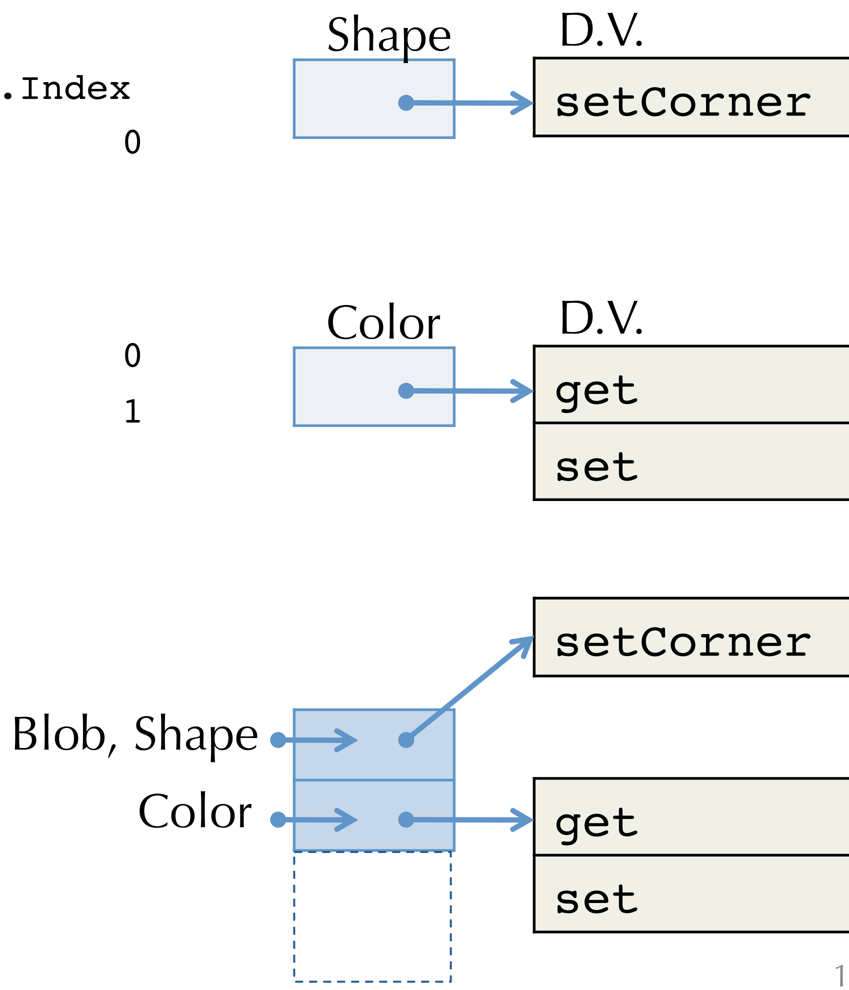
Option 3: Multiple Dispatch Vectors

- 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.

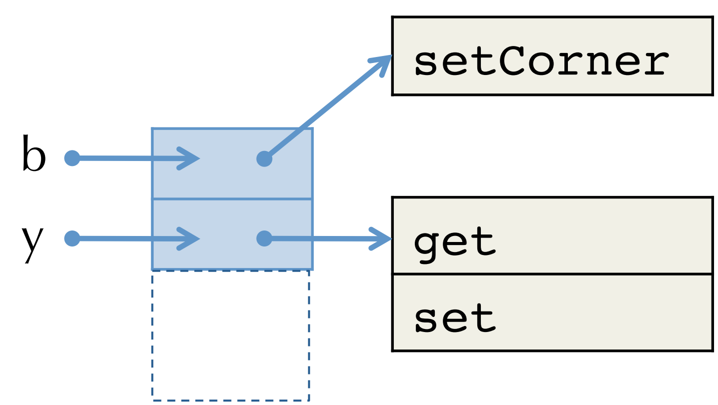
```
Blob b = new Blob();
Color y = b;     // implicit cast!
```

- 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 of fields

Static fields and methods

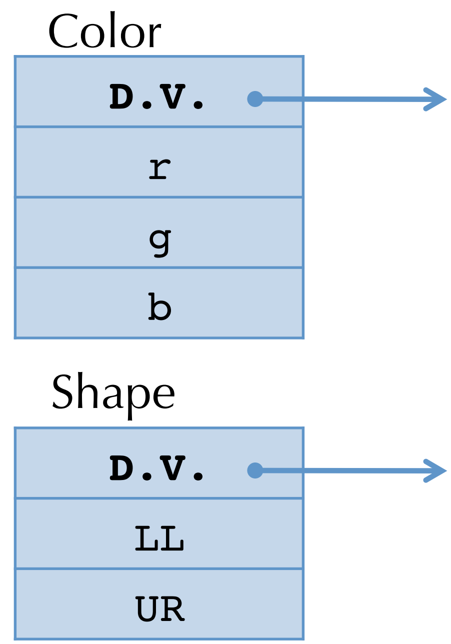
Comparison with closures

OTHER CONSIDERATIONS

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.

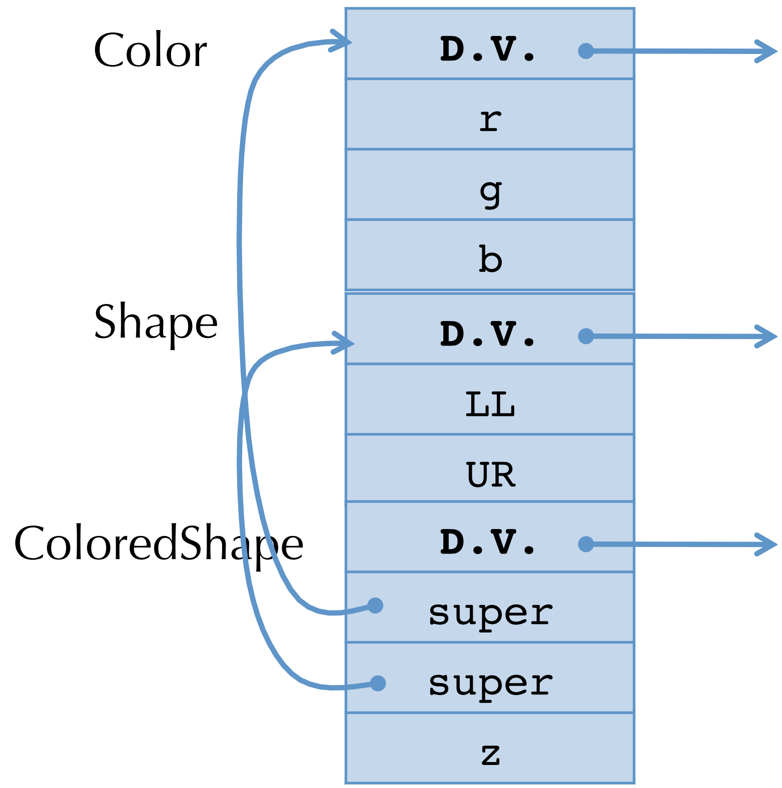
```
class Color {
  float r, g, b; /* offsets: 4,8,12 */
}
class Shape {
  Point LL, UR; /* offsets: 4, 8 */
}
class ColoredShape extends
Color, Shape {
  int z;
}
```



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 and fields
  - Static methods and fields belong to a class, not the instances of the class.
  - Storage is allocated with the dispatch vectors
  - Static methods have no “this” parameter (no receiver object)

- `A.m()` and `A.f` compute the address of `A`’s vtable to access `m` and `f`

- Methods are 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)

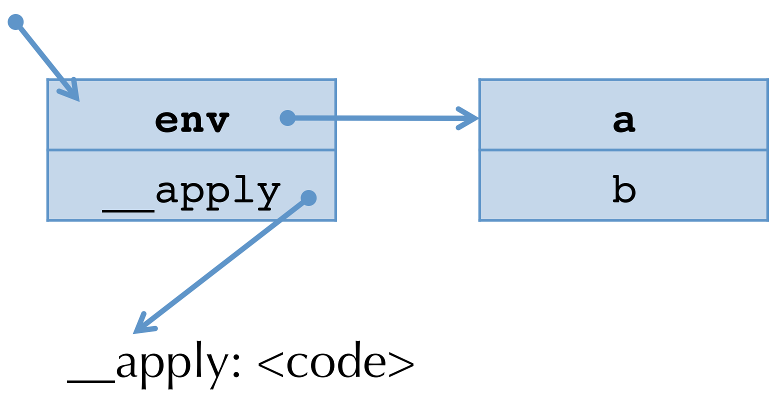
Compiling Constructors

- 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.
  - There are issues with consistency and typechecking

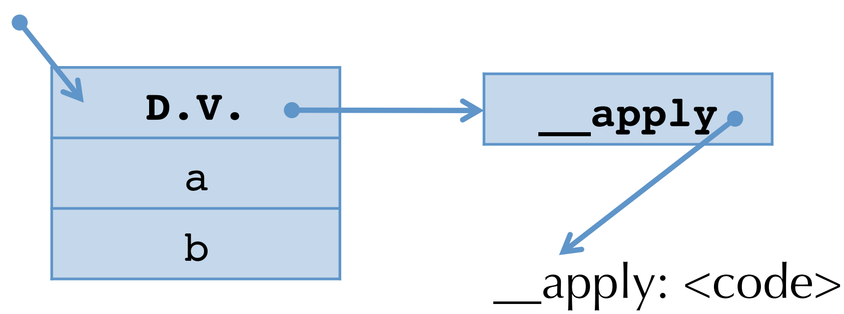
Observe: Closure ≈ Single-method Object

- 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
  }
}
```



See oat.pdf (Project 5 version)

TYPECHECKING CLASSES