

CS 4120

Introduction to Compilers

Andrew Myers
Cornell University

Lecture 31: Object layout and
method dispatch

18 Apr 2016

Classes

- Components
 - fields/instance variables
 - values may differ from object to object
 - usually mutable
 - methods
 - values shared by all objects of a class
 - inherited from superclasses
 - usually immutable
 - usually function values with implicit argument: object itself (this/self)
 - all components have visibility: public/private/protected (subclass visible)

Implementing classes

- Environment binds type names to type objects, *i.e. class objects*
 - Java: class object visible in programming language (`java.lang.Class`)
- Class objects are environments:
 - identifier bound to type
 - + expression (e.g. method body)
 - + field/method
 - + static/non-static
 - + visibility

Code generation for objects

- Methods
 - Generating method code
 - Generating method calls (dispatching)
- Fields
 - Memory layout
 - Generating accessor code
 - Packing and alignment

Compiling methods

- Methods look like functions, are type-checked like functions...what is different?
- Argument list: implicit receiver argument
- Calling sequence: use *dispatch vector*

The need for dispatching

- Problem: compiler can't tell what code to run when method is called

```
interface Point { int getx(); float norm(); }
```

```
class ColoredPoint implements Point {...
```

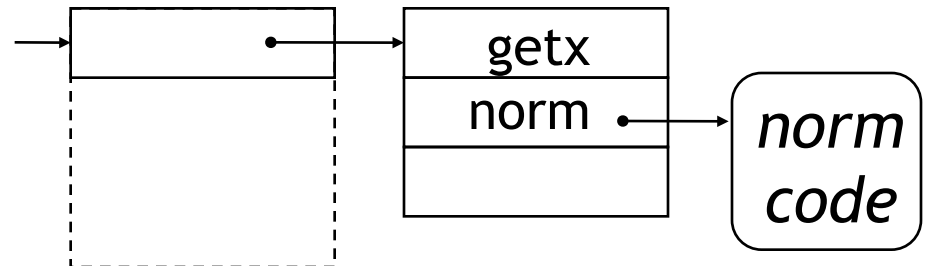
```
    float norm() { return sqrt(x*x+y*y); }
```

```
class 3DPoint implements Point { ...
```

```
    float norm() return sqrt(x*x+y*y+z*z); }
```

- Solution: dispatch

table (dispatch
vector, selector table...)



Method dispatch

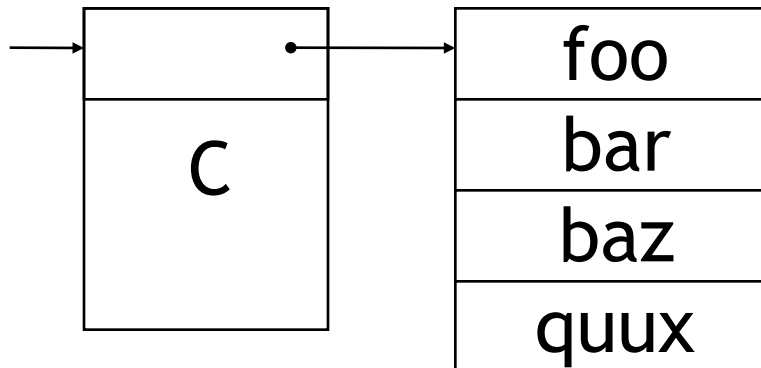
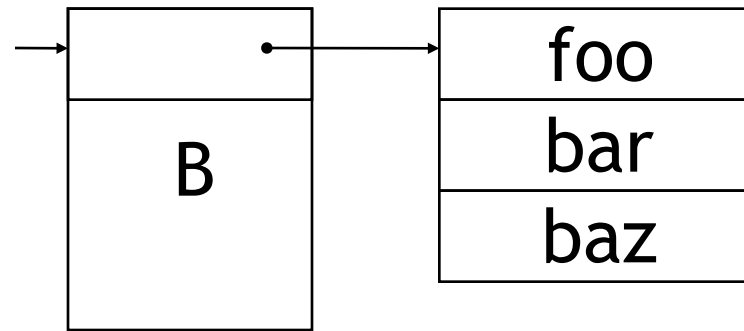
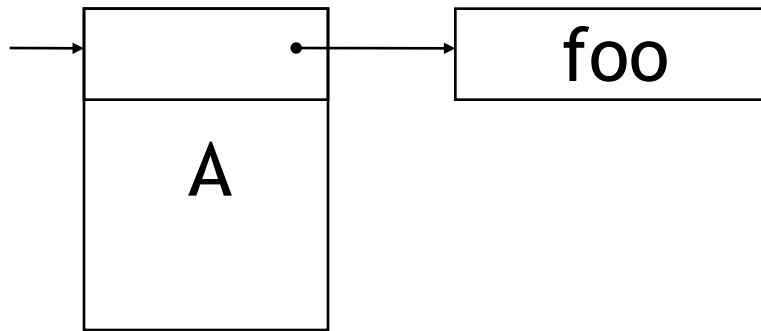
- Idea: every method has its own small integer index
- Index is used to look up method in dispatch vector

```
interface A {  
    void foo();  
}  
interface B extends A {  
    void bar();  
    void baz();  
}
```

```
class C implements B {  
    void foo() {...}  
    void bar() {...}  
    void baz() {...}  
    void quux() {...}  
}
```

A
|
B
|
C

Dispatch vector layouts

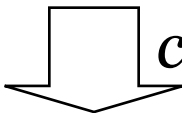


A	foo
B	bar, baz
C	quux

Method arguments

- Methods have a special variable (in Java, “this”) called the *receiver object*
- Historically (Smalltalk): method calls thought of as *messages* sent to *receivers*
- Receiver object is (implicit) argument to method

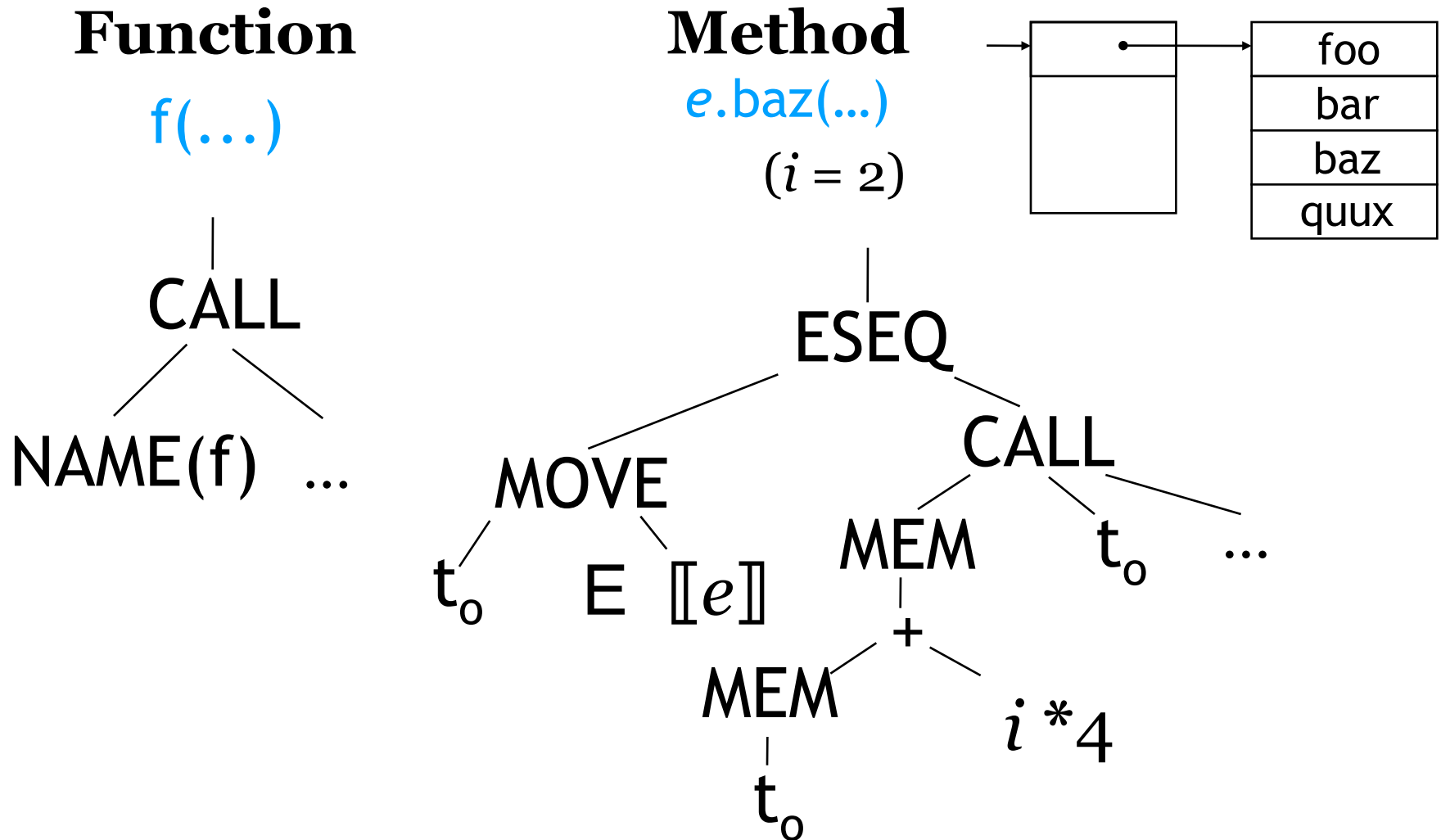
```
class Shape {  
    int setCorner(int which, Point p) { ... }  
}
```

 *compiled like*

```
int setCorner(Shape this, int which, Point p) { ... }
```

How do we know the type of “this”?

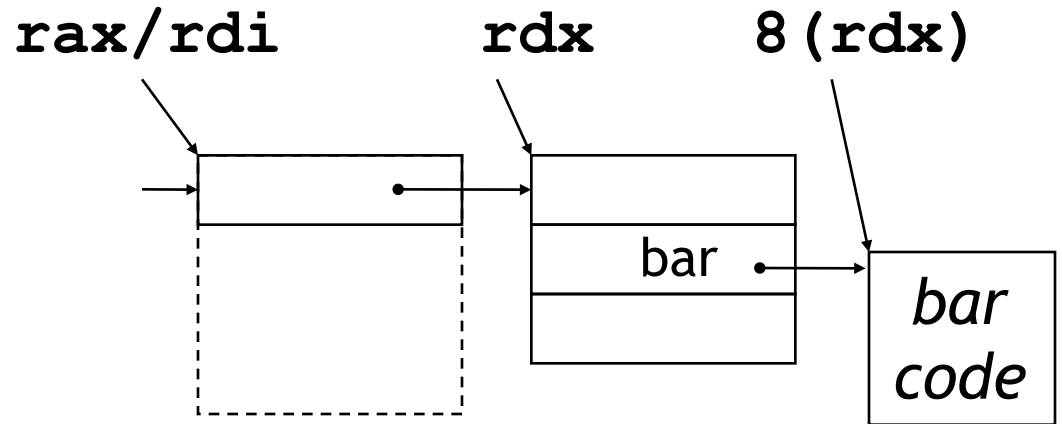
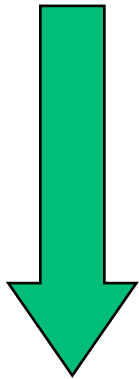
Calling sequence



Example

A	foo
B	bar,baz
C	quux

b.bar(3);



```
movq  (%rax) , %rdx  (get DV)
movq  %rax, %rdi      (arg 1)
movq  $3, %rsi        (arg 2)
callq *8(%rdx)        (bar=1)
```

Inheritance

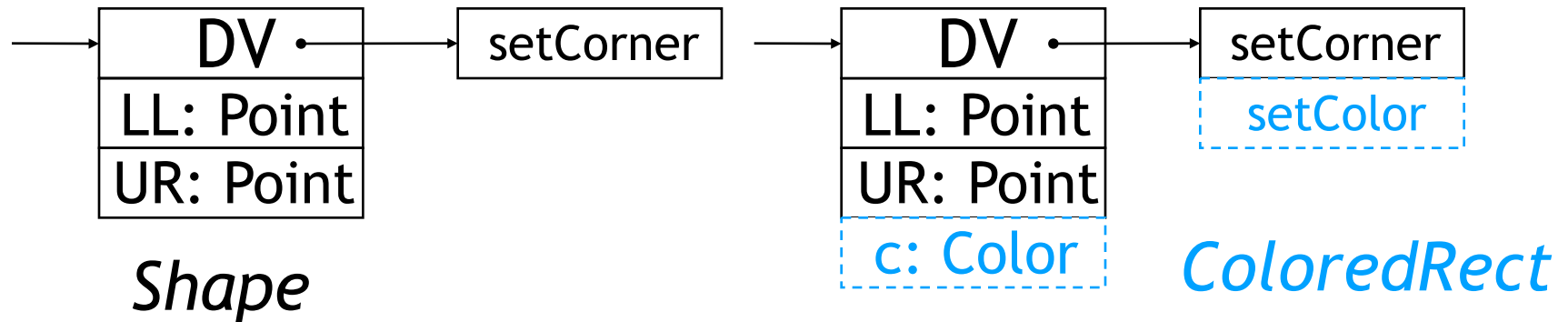
- Three traditional components of object-oriented languages
 - abstraction/encapsulation/information hiding
 - subtyping/interface inheritance -- interfaces inherit method signatures from supertypes
 - inheritance/implementation inheritance -- a class inherits signatures *and* code from a superclass (possibly “abstract”)

Inheritance

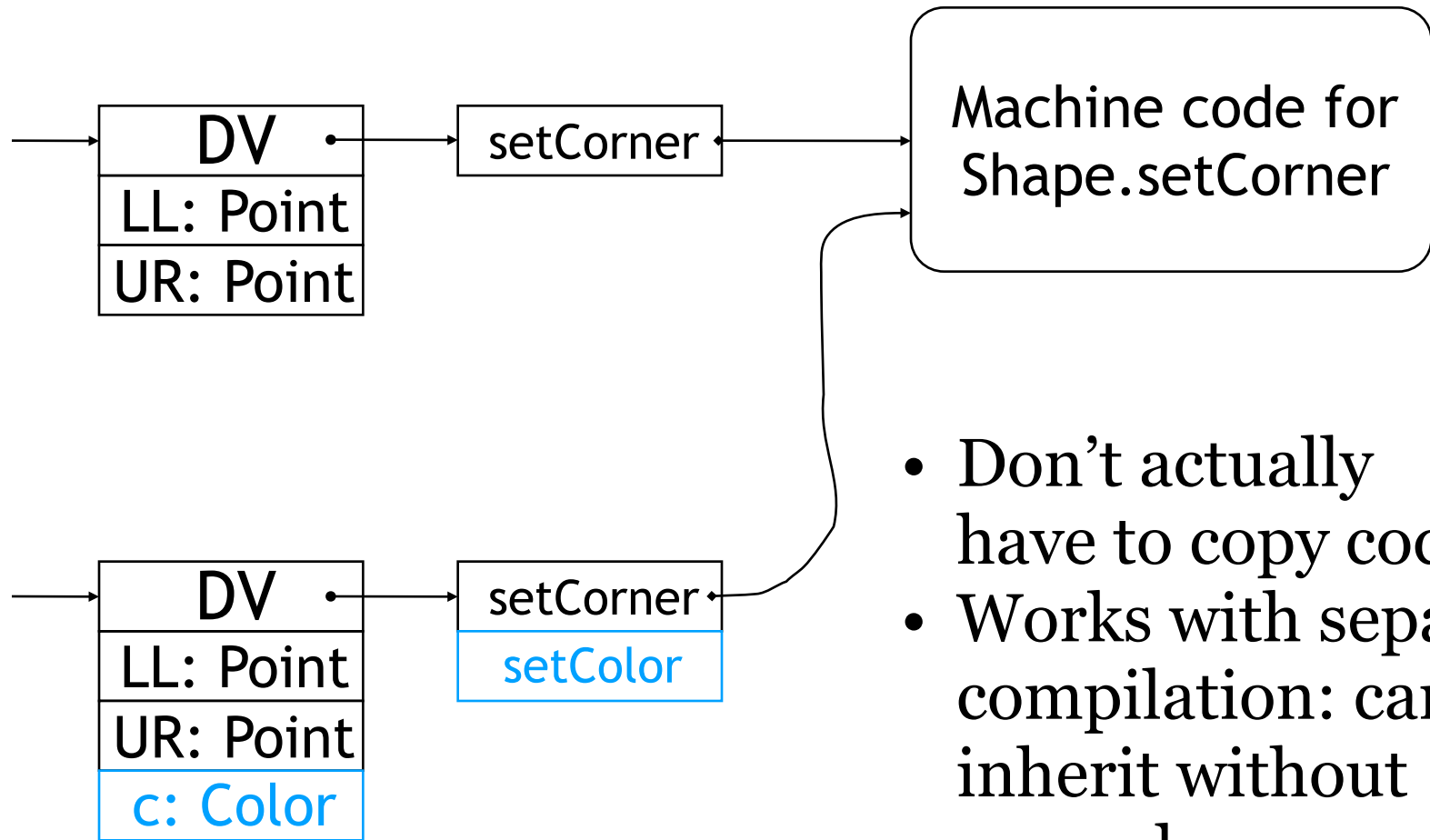
- Method code copied down from superclass if not *overridden* by subclass
- Fields also inherited (needed by inherited code in general)
- Fields checked just as for records: mutable fields must be invariant, immutable fields may be covariant

Object Layout

```
class Shape {  
    Point LL, UR;  
    void setCorner(int which, Point p);  
}  
class ColoredRect extends Shape {  
    Color c;  
    void setColor(Color c_);  
}
```



Code Sharing



- Don't actually have to copy code!
- Works with separate compilation: can inherit without superclass source

Interfaces, abstract classes

- Classes define a type *and* some values (methods)
- Interfaces are pure object types : no implementation
 - no dispatch vector: only a DV layout
- Abstract classes are halfway:
 - define some methods
 - leave others unimplemented
 - no objects (instances) of abstract class
- DV needed only for real classes

Static methods

- In Java, can declare methods *static* -- they have no receiver object
- Called exactly like normal functions
 - don't need to enter into dispatch vector
 - don't need implicit extra argument for receiver
- Treated as methods as way of getting functions inside the class scope (access to module internals for semantic analysis)

Constructors

- Java, C++: classes can declare *object constructors* that create new objects:
`new C(x, y, z)`
- Other languages (Modula-3, Iota⁺): objects constructed by “new C”; no initialization code

```
class LenList {  
    int len, head; List next;  
    LenList() { len = 0; }  
}
```

Compiling constructors

- Compiled just like static methods except:
 - pseudo-variable “this” is in scope as in methods
 - this is initialized with newly allocated memory
 - first word in memory initialized to point to DV
 - value of this is return value of code

LenList() { len = 0; }

LenList\$constr:
movq \$0, 8(%rdi)
ret

...
movq \$16, %rdi ; 3 fields
callq GC_malloc
movq \$LenList_DV, (%rax)
movq %rax, %rdi
callq LenList\$constructor

_DATA SEGMENT
LenList_DV DWORD LenList\$first
 DWORD LenList\$rest
 DWORD LenList\$length
_DATA ENDS