COMP 412
FALL 2010

The Procedure Abstraction, Part VI: Inheritance in OOLs

Comp 412

What About Inheritance?

Impact on OR Layout

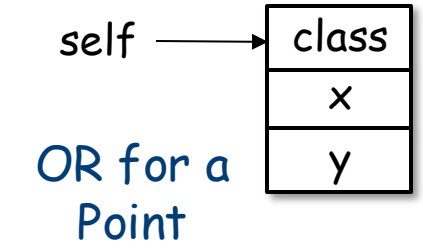
Assume single inheritance

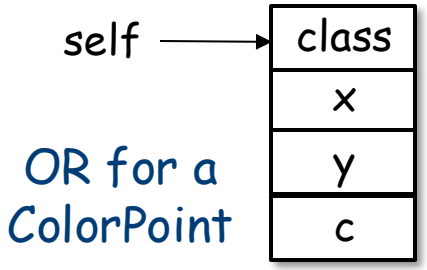
- OR needs slots for each member declared, all the way up the class hierarchy (class, superclass, super-superclass, …)
- Can use prefixing of storage to lay out the OR

Back to Our Java Example — Class Point

```
Class Point {
    public int x, y;
    …
}

Class ColorPoint extends Point {
    Color c;
    …
}
```





What happens if we cast a ColorPoint to a Point?

Take the word extends literally.

Open World versus Closed World

Prefixing assumes that the class structure is known when layout is performed. Two common cases occur.

Closed-World Assumption (Compile time)

- Class structure is known and closed prior to runtime
- Can lay out ORs in the compiler and/or the linker

Open-World Assumption (Interpreter or JIT)

- Class structure can change at runtime
- Cannot lay out ORs until they are allocated
  - Walk class hierarchy at allocation

C++ has a closed class structure.

Java as an open class structure.

Open World versus Closed World

What happens if the class structure changes for a class with active instantiated objects?

- Oops. That might be a problem.

Changes to the structure of an instantiated class

- Languages differ on legality and advisability of such changes
- Smalltalk-80, for example, found & fixed all ORs in the class
- Python does not support such changes
  - *Programmer can, on her own, track instances and re-instantiate them …*

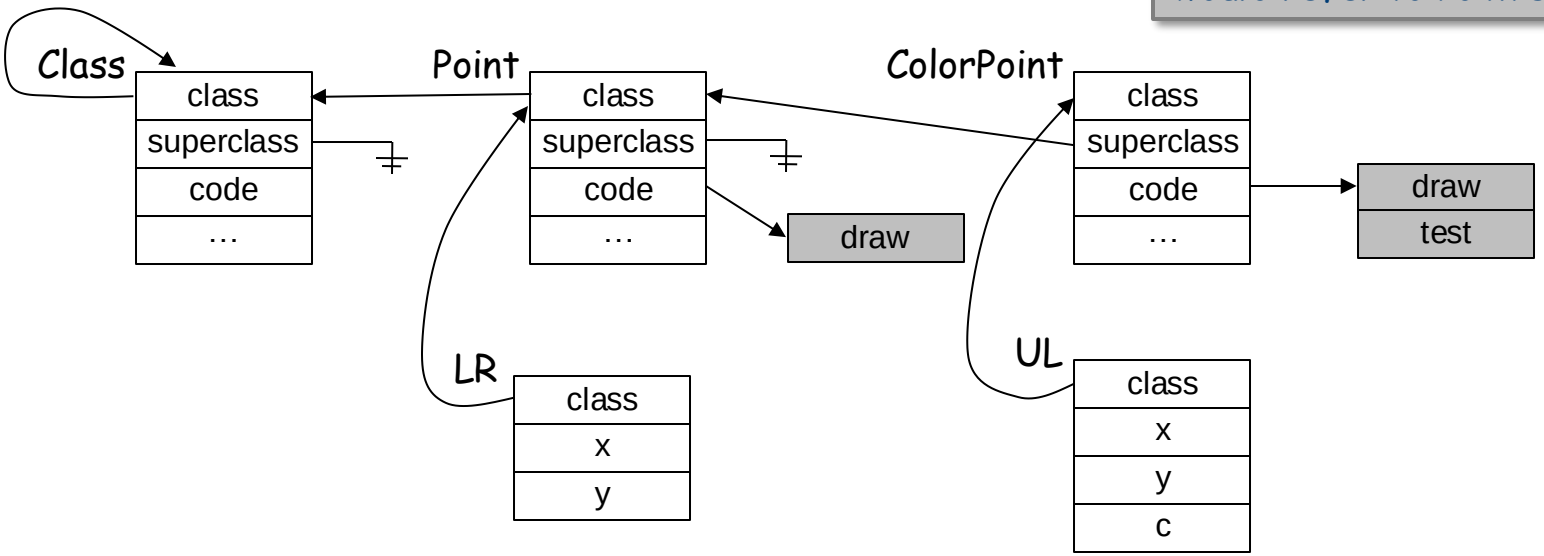
What About Code Members?

How does the language's runtime environment find the code for a given method invocation?

Closed Class Structure

- Mapping of names to methods is static and known (C++)
  - Fixed offsets & indirect calls
- Virtual functions force runtime resolution

If ColorPoint inherited draw from Point, its code vector would refer to Point's draw.



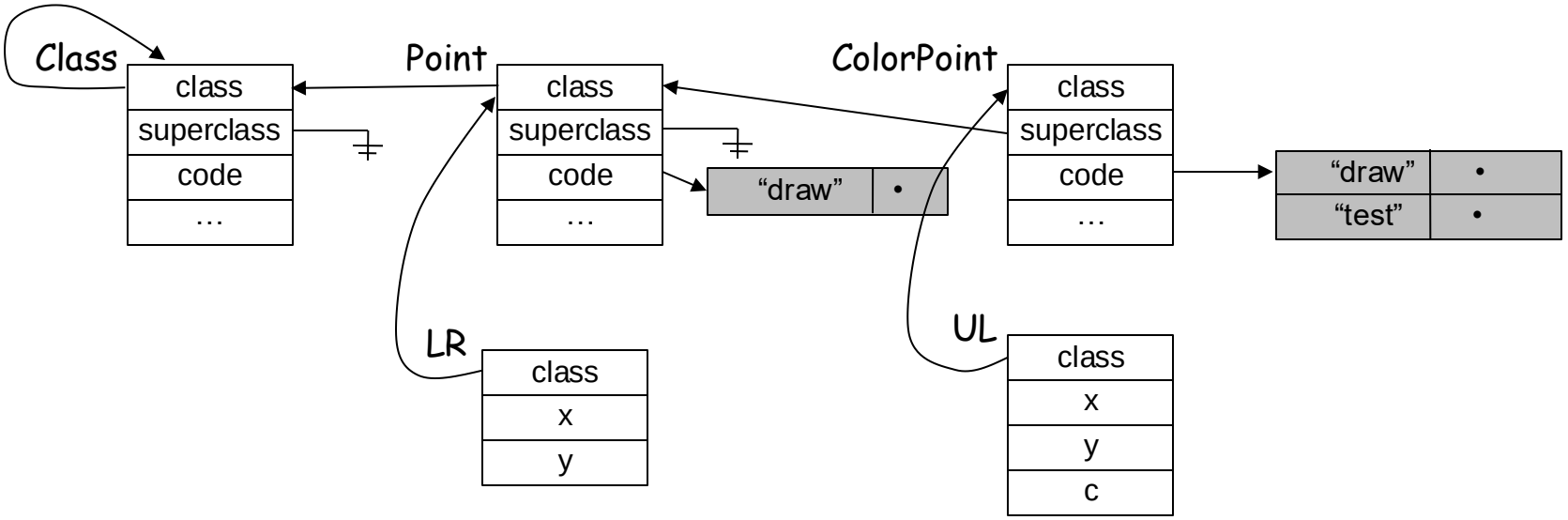
UL finds draw at offset 0 in ColorPoint's code vector

What About Code Members?

How does the language's runtime environment find the code for a given method invocation?

Open Class Structure

- Dynamic mapping, unknown until runtime
- In general case, need runtime representation of hierarchy
  - Lookup by textual name in class' table of methods



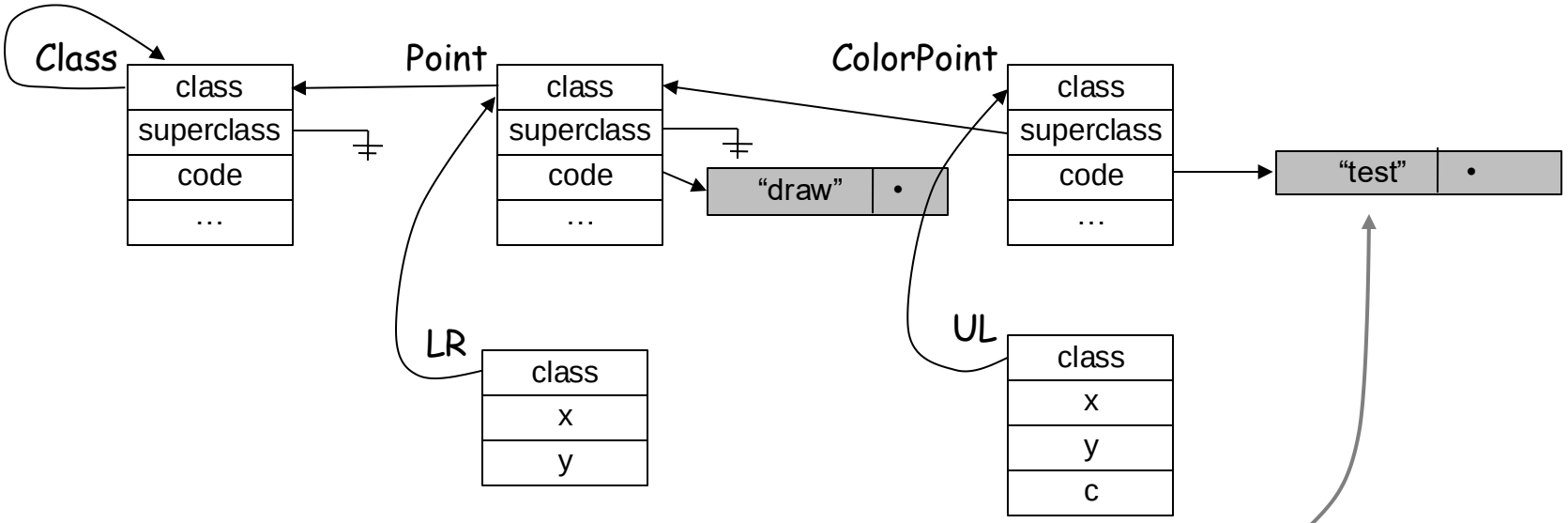
UL finds draw at offset 0 in ColorPoint's code vector

What About Code Members?

How does the language's runtime environment find the code for a given method invocation?

Open Class Structure

- Dynamic mapping, unknown until runtime
- In general case, need runtime representation of hierarchy
  - Lookup by textual name in class' table of methods



If ColorPoint inherited draw from Point, its code vector would lack a draw.

The Single Inheritance Hierarchy

Two distinct philosophies

Closed Class Structure (C++)
- Can map name to code at compile time
- Leads to 1-level code vector
- Copy superclass methods
- Fixed offsets & indirect calls
- Less flexible & expressive

Open Class Structure (ST80)
- Cannot map name to code at compile time
- Multiple jump vectors (1/class)
- Must search for method
- Run-time lookup caching
- Much more expensive to run

Impact of OOL on Program's Name Space
- Method can see values in the code's lexical hierarchy
- Method can see members of self, class, & superclasses
- Many different levels where a value can reside

OOL differs from ALL in the shape of its name space ***AND*** in the mechanism used to bind names to implementations

The Single Inheritance Hierarchy

Two distinct philosophies

Closed Class Structure (C++)

- Can map name … compile time
- Leads to 1-lev…
- Copy supercl…
- Fixed offsets…
- Less flexible…

Runtime changes (Java)

- Layout OR by prefixing
- Limit changes in instantiated classes
- Use 1-level code vectors and rebuild on hierarchy change
- Fixed offsets & 1-level of indirection on calls

Open Class Structure (ST80)

- …ame to code at …
- … vectors (1/class)
- …for method …
- …up caching
- …xpensive to run

Impact of OOL on Program's Name Space

- Method can see values in the code's lexical hierarchy
- Method can see members of self, class, & superclasses
- Many different levels where a value can reside

OOL differs from ALL in the shape of its name space ***AND*** in the mechanism used to bind names to implementations

What About Multiple Inheritance?

The Idea

- Let *C* be a subclass of both *A* and *B*
- *C* draws some, but not all, methods from *A*
- *C* draws some, but not all, methods from *B*

Need a linguistic mechanism to specify partial inheritance

Problems arise when *C* inherits from both *A* & *B*

- *C*'s OR can extend *A* or *B*, but not both
- *C*'s code vector can extend *A* or *B*, but not both
- Some class *D* might inherit from either *A* or *B* or both
  - Need consistency in OR layout & code vector layout
- Both *A* & *B* might provide fum() — which is seen in *C*?
  - C++ produces a "syntax error" when fum() is used

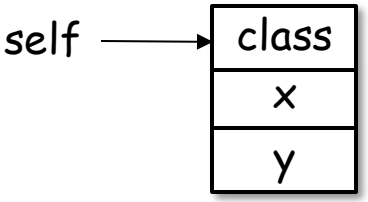
Need a better way to say "inherit"

Modified example

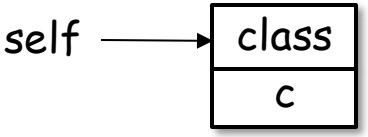
OR Layout with Multiple Inheritance

We can try to use prefixing again

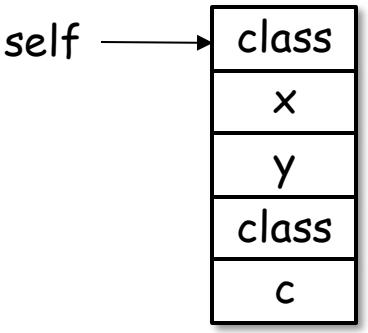
```
Class Point {
      public int x, y;
      public void draw();
      public Point add();
}
```




```
Class CThing {
      Color c;
      public void colorize();
}
```




```
Class CPoint extends
    Point  and CThing {
    public void draw();
}
```



How do casts work?

- If we cast a CPoint as a Point, x & y are at the desired offsets
- If we cast a CPoint as a Cthing, c is at the wrong offset
  - Extra class field is the key
  - Cast bumps "self" to point at 2nd class field

Class that occurs out of "single inheritance prefix order" needs additional information – class field.

Assume 64 bit slots in the ORs

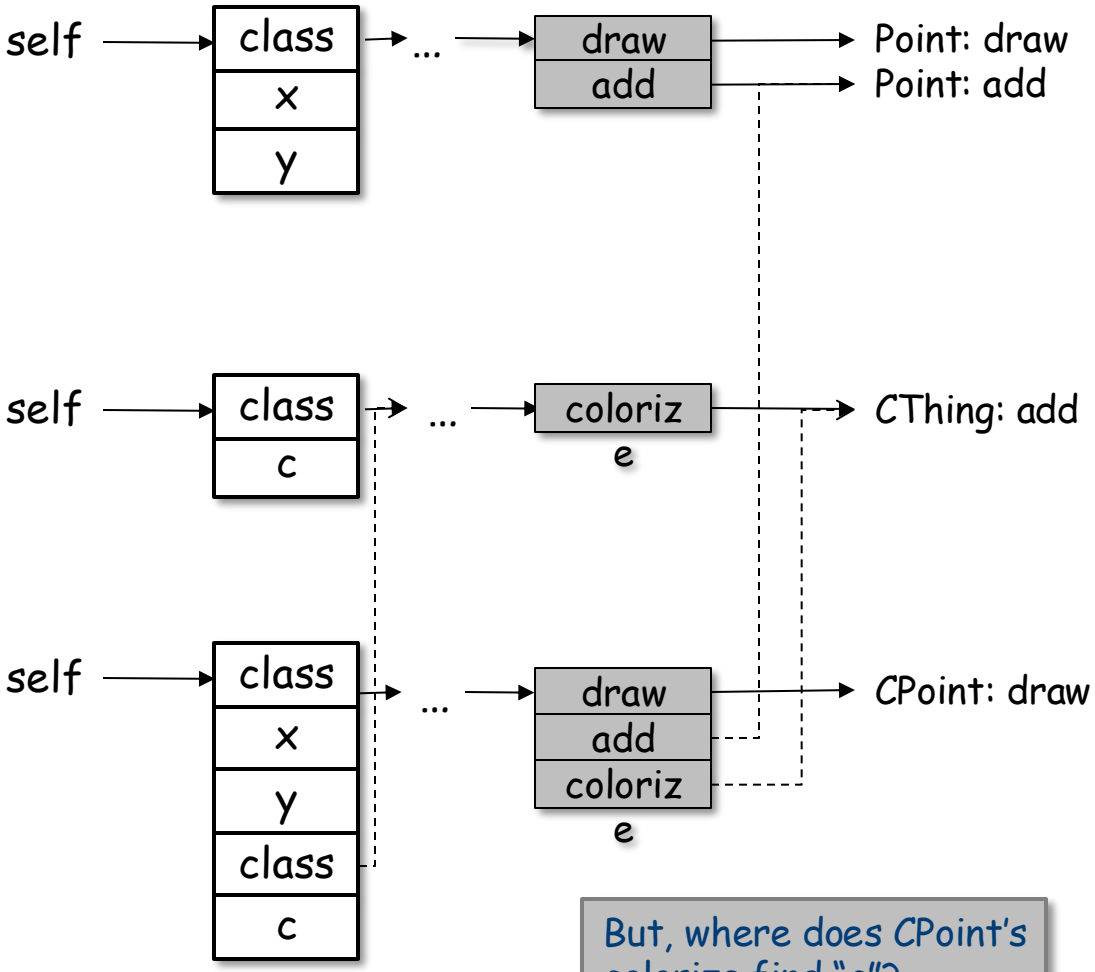
OR Layout with Multiple Inheritance

Need to prefix both OR & code vector

```
Class Point {
      public int x, y;
      public void draw();
      public Point add();
}

Class CThing {
      Color c;
      public void colorize();
}

Class CPoint extends
    Point  and CThing {
    public void draw();
}
```



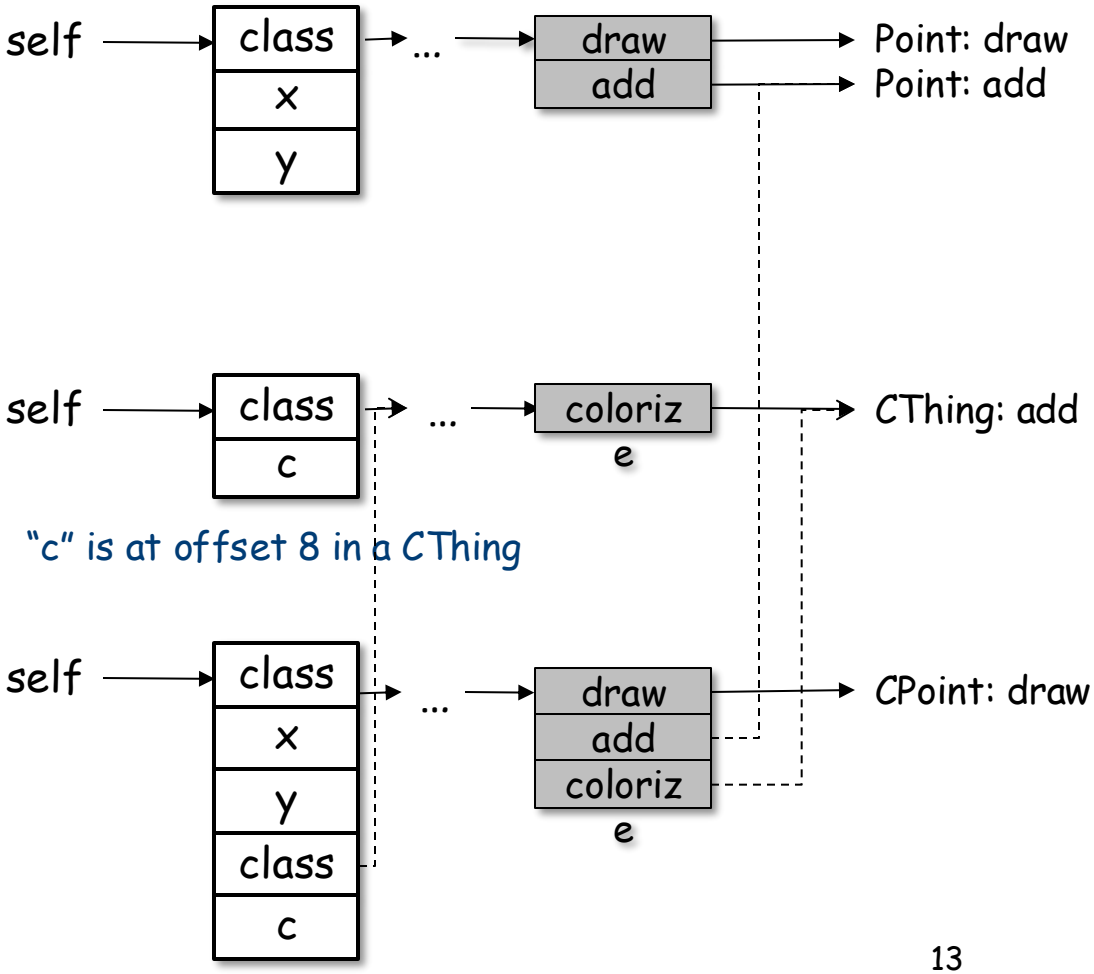
OR Layout with Multiple Inheritance

Solution: Prefix both code storage & code vector

```
Class Point {
      public int x, y;
      public void draw();
      public Point add();
}

Class CThing {
      Color c;
      public void colorize();
}

Class CPoint extends
    Point  and CThing {
    public void draw();
}
```



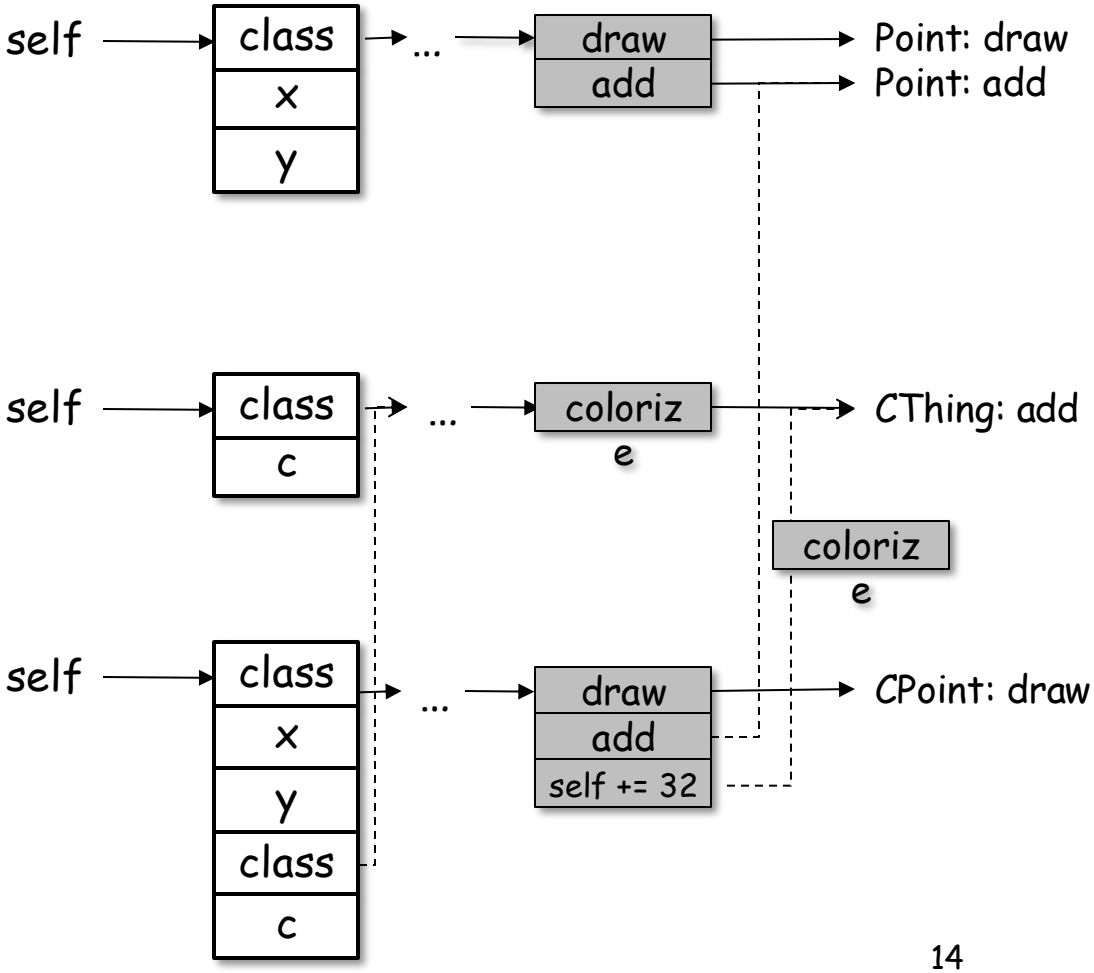
OR Layout with Multiple Inheritance

To fix CPoint's offset for "c", use a trampoline function

```
Class Point {
      public int x, y;
      public void draw();
      public Point add();
}

Class CThing {
      Color c;
      public void colorize();
}

Class CPoint extends
    Point  and CThing {
    public void draw();
}
```



Open World versus Closed World

Closed Class Structure

- Compile time or link time layout (ORs, classes, code vectors)
- If known at compile time, can generate static code
  - No code vector, no indirection, efficient implementation
- C++ virtual functions use runtime resolution, as if open world

Open Class Structure

- Cannot lay out ORs until hierarchy is known
- With infrequent change, can perform layout at each change
  - Single-level code vectors, fixed offsets, indirect calls
- With frequent change, may need runtime method resolution
  - Search for method in class hierarchy (w/tag) & cache result
  - Much more expensive

Method Calls

Given the runtime structure, how does a call work?

- Compiled code does not contain the callee's address
- Reference is relative to receiver
  - Through class to code vector

In the general case, may need dynamic dispatch

- Map code member to a search key
- Perform runtime search through hieraarchy
  - This process is expensive
- Use a "method cache" to speed the search
  - Cache holds <*search key, class, method pointer*>

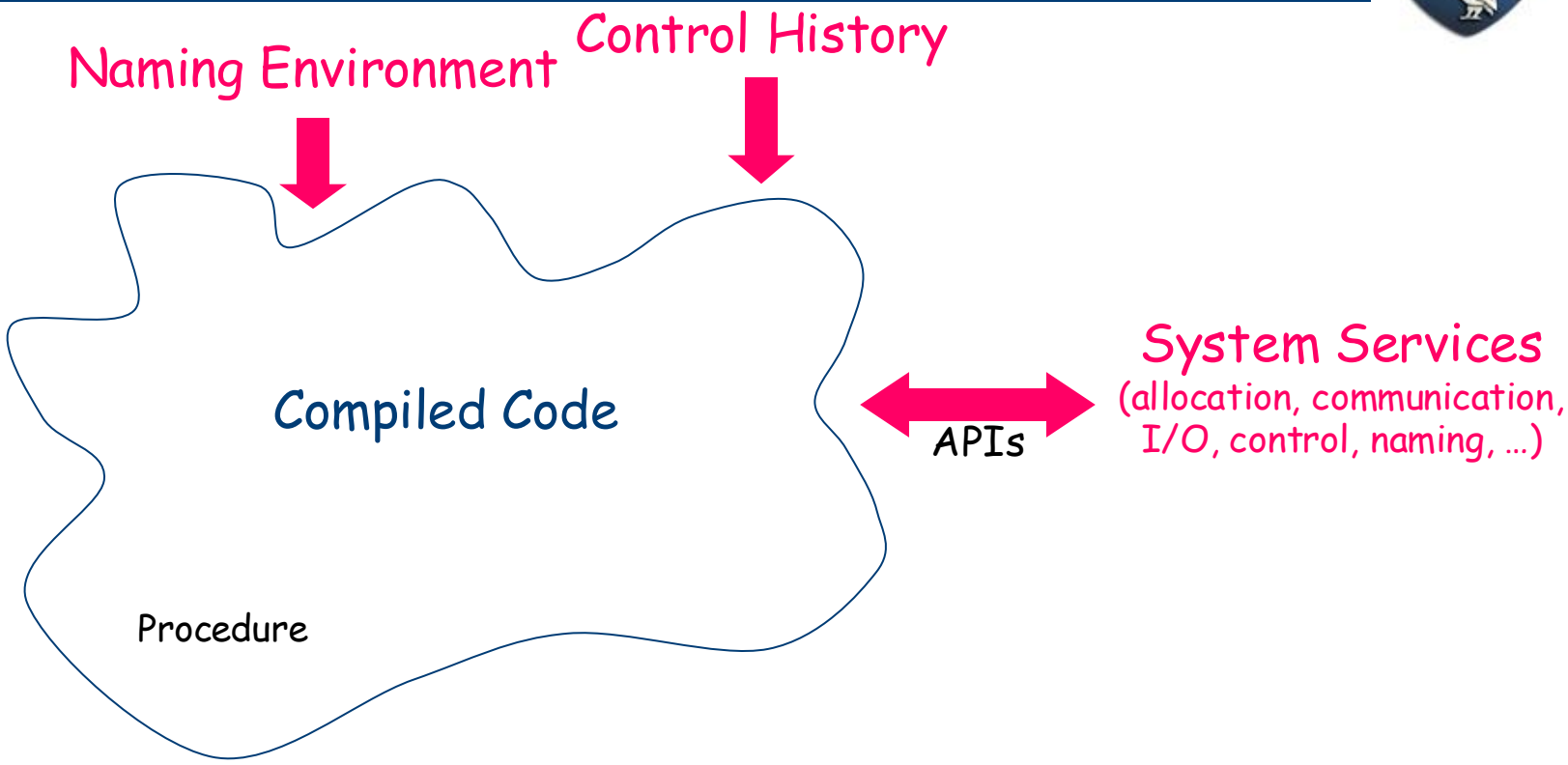
Smalltalk-80

Method Calls

Improvements are possible in special cases

- If class has no subclasses, can generate direct calls
  - Class structure must be static or class must be FINAL
- If class structure is static — Language design
  - Can generate complete method table for each class
    - → Use prefixed object records and complete code vectors
  - Indirection through the class point
  - Keeps overhead low
- If class structure changes infrequently — Behavior
  - Build complete method tables at initialization & when class structure changes
- If running program can create new classes — Language design
  - Well, not all things can be efficient
  - See Deutsch & Schiffman, POPL 1984

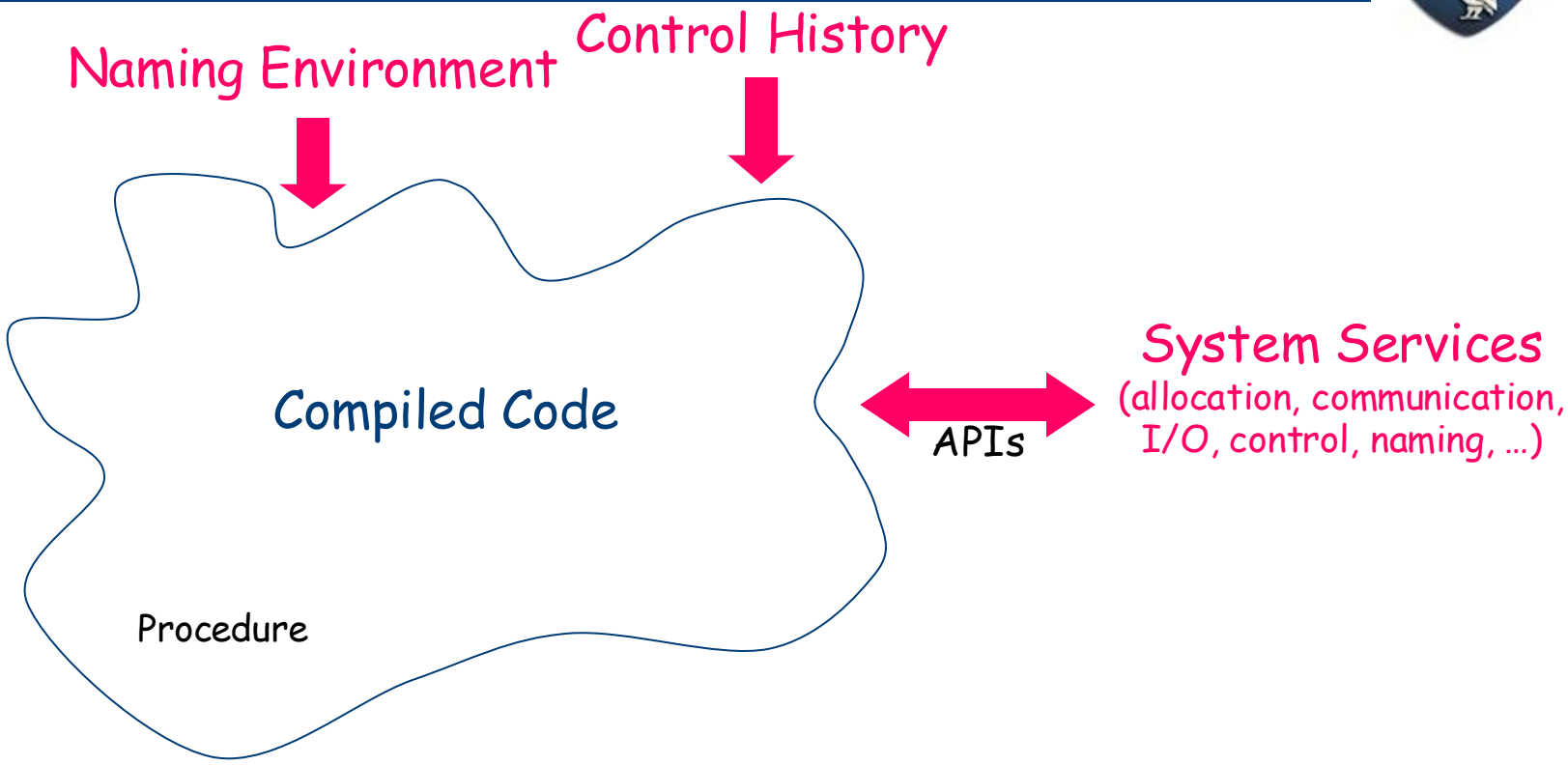
The Procedure & Its Three Abstractions



We looked at the three abstractions for ALLs and OOLs

- OOLs have more complex naming environments than ALLs
- Need both compile time and run time support for naming

The Procedure & Its Three Abstractions



Procedure invocation reflects these differences

- OOLs use indirect calls through a code vector or jump table
- Extra loads at each call to engineer name resolution