

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Multiple Inheritance

What next?

Have used classes for OOP's essence: inheritance, overriding, dynamic dispatch

Now, what if we want to have more than *just 1 superclass*

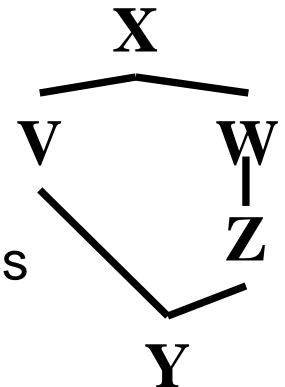
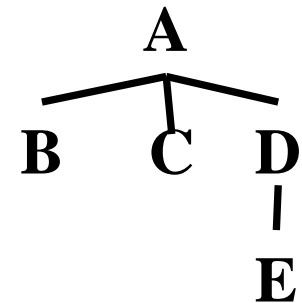
- *Multiple inheritance*: allow > 1 superclasses
 - Useful but has some problems (see C++)
- Ruby-style *mixins*: 1 superclass; > 1 method providers
 - Often a fine substitute for multiple inheritance and has fewer problems (see also Scala *traits*)
- Java/C#-style *interfaces*: allow > 1 types
 - Mostly irrelevant in a dynamically typed language, but fewer problems

Multiple Inheritance

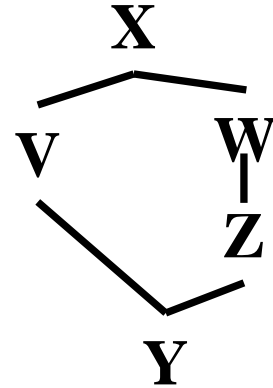
- If inheritance and overriding are so useful, why limit ourselves to one superclass?
 - Because the semantics is often awkward (this topic)
 - Because it makes static type-checking harder (not discussed)
 - Because it makes efficient implementation harder (not discussed)
- Is it useful? Sure!
 - Example: Make a **ColorPt3D** by inheriting from **Pt3D** and **ColorPt** (or maybe just from **Color**)
 - Example: Make a **StudentAthlete** by inheriting from **Student** and **Athlete**
 - With single inheritance, end up copying code or using non-OOP-style helper methods

Trees, dags, and diamonds

- Note: The phrases *subclass*, *superclass* can be ambiguous
 - There are *immediate* subclasses, superclasses
 - And there are *transitive* subclasses, superclasses
- Single inheritance: the *class hierarchy* is a tree
 - Nodes are classes
 - Parent is immediate superclass
 - Any number of children allowed
- Multiple inheritance: the class hierarchy no longer a tree
 - Cycles still disallowed (a directed-acyclic graph)
 - If multiple paths show that *X* is a (transitive) superclass of *Y*, then we have *diamonds*



What could go wrong?



- If *V* and *Z* both define a method **m**, what does *Y* inherit? What does **super** mean?
 - *Directed resends* useful (e.g., **z : : super**)
- What if *X* defines a method **m** that *Z* but not *V* overrides?
 - Can handle like previous case, but sometimes undesirable (e.g., **ColorPt3D** wants **Pt3D**'s overrides to “win”)
- If *X* defines fields, should *Y* have one copy of them (**f**) or two (**v : : f** and **z : : f**)?
 - Turns out each behavior can be desirable (next slides)
 - So C++ has (at least) two forms of inheritance

3DColorPoints

If Ruby had multiple inheritance, we would want `ColorPt3D` to inherit methods that share one `@x` and one `@y`

```
class Pt
  attr_accessor :x, :y
  ...
end
class ColorPt < Pt
  attr_accessor :color
  ...
end
class Pt3D < Pt
  attr_accessor :z
  ... # override some methods
end
class ColorPt3D < Pt3D, ColorPt # not Ruby!
end
```

ArtistCowboys

This code has `Person` define a pocket for subclasses to use, but an `ArtistCowboy` wants *two* pockets, one for each `draw` method

```
class Person
  attr_accessor :pocket
  ...
end
class Artist < Person # pocket for brush objects
  def draw # access pocket
  ...
end
class Cowboy < Person # pocket for gun objects
  def draw # access pocket
  ...
end
class ArtistCowboy < Artist, Cowboy # not Ruby!
end
```