CSE341: Programming Languages
Lecture 22
Multiple Inheritance, Interfaces, Mixins

Dan Grossman
Fall 2011
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++)

• Java-style *interfaces*: allow > 1 types
  – Mostly irrelevant in a dynamically typed language, but fewer problems

• Ruby-style *mixins*: 1 superclass; > 1 method providers
  – Often a fine substitute for multiple inheritance and has fewer problems
Multiple Inheritance

• If inheritance and overriding are so useful, why limit ourselves to one superclass?
  – Because the semantics is often awkward (next couple slides)
  – 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 \textit{subclass}, \textit{superclass} can be ambiguous
  - There are \textit{immediate} subclasses, superclasses
  - And there are \textit{transitive} subclasses, superclasses

- Single inheritance: the \textit{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 \textit{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 is sometimes 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

```ruby
class Pt
  attr_accessor :x, :y
  ...
end

class ColorPt < Pt
  attr_accessor :color
  ...
end

class Pt3D < Pt
  attr_accessor :z
  ... # override methods like distance?
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.

```ruby
class Person
  attr_accessor :pocket
  ...
end

class Artist < Person # pocket for brush objects
  def draw # access pocket
    ...
  end
end

class Cowboy < Person # pocket for gun objects
  def draw # access pocket
    ...
  end
end

class ArtistCowboy < Artist, Cowboy # not Ruby!
end
```
Java interfaces

Recall (?), Java lets us define *interfaces* that classes explicitly *implement*

```java
interface Example {
    void m1(int x, int y);
    Object m2(Example x, String y);
}

class A implements Example {
    public void m1(int x, int y) {...}
    public Object m2(Example e, String s) {...}
}
class B implements Example {
    public void m1(int pizza, int beer) {...}
    public Object m2(Example e, String s) {...}
}
```
What is an interface?

interface Example {
    void m1(int x, int y);
    Object m2(Example x, String y);
}

• An interface is a type!
  – Any implementer (including subclasses) is a subtype of it
  – Can use an interface name wherever a type appears
  – (In Java, classes are also types in addition to being classes)
• An implementer type-checks if it defines the methods as required
  – Parameter names irrelevant to type-checking; it's a bit strange that Java requires them in interface definitions
• A user of type Example can objects with that type have the methods promised
  – I.e., sending messages with appropriate arguments type-checks
Multiple interfaces

• Java classes can implement any number of interfaces

• Because interfaces provide no methods or fields, no questions of method/field duplication arise
  – No problem if two interfaces both require of implementers and promise to clients the same method

• Such interfaces aren't much use in a dynamically typed language
  – We don't type-check implementers
  – We already allow clients to send any message
  – Presumably these types would change the meaning of is_a?, but we can just use instance_methods to find out what methods an object has
Why no interfaces in C++?

If you have multiple inheritance and abstract methods (called pure virtual methods in C++), there is no need for interfaces

- **Abstract method**: A method declared but not defined in a class. All instances of the (sub)class must have a definition

- **Abstract class**: Has one or more abstract methods; so disallow creating instances of this exact class
  - Have to subclass and implement all the abstract methods to create instances

- Little point to abstract methods in a dynamically typed language

- In C++, instead of an interface, make a class with all abstract methods and inherit from it – same effect on type-checking
Mixins

- A *mixin* is (just) a collection of methods
  - Less than a class: no fields, constructors, instances, etc.
  - More than an interface: methods have bodies

- Languages with mixins (e.g., Ruby modules) typically allow a class to have one superclass but any number of mixins

- Semantics: *Including a mixin makes its methods part of the class*
  - Extending or overriding in the order mixins are included in the class definition
  - More powerful than helper methods because mixin methods can access methods (and instance variables) on `self` not defined in the mixin
Example

```ruby
module Doubler
  def double
    self + self # assume included in classes w/ +
  end
end

class String
  include Doubler
end

class AnotherPt
  attr_accessor :x, :y
  include Doubler
  def + other
    ans = AnotherPt.new
    ans.x = self.x + other.x
    ans.y = self.y + other.y
    ans
  end
end
```
Lookup rules

Mixins change our lookup rules slightly:

- When looking for receiver \texttt{obj0}'s method \texttt{m}, look in \texttt{obj0}'s class, then mixins that class includes (later includes shadow), then \texttt{obj0}'s superclass, then the superclass' mixins, etc.

- As for instance variables, the mixin methods are included in the same object
  - So usually bad style for mixin methods to use instance variables since a name clash would be like our \texttt{CowboyArtist} pocket problem (but sometimes unavoidable?)
The two big ones

The two most popular/useful mixins in Ruby:

• Comparable: Defines <, >, ==, !=, >=, <= in terms of <=>

• Enumerable: Defines many iterators (e.g., map, find) in terms of each

Great examples of using mixins:
  – Classes including them get a bunch of methods for just a little work
  – Classes do not "waste" their "one superclass" for this
  – Do not need the complexity of multiple inheritance

• See lec22.rb for some example uses
Replacement for multiple inheritance?

• A mixin probably works well for ColorPt3D:
  – Color a reasonable mixin except for using an instance variable

```ruby
module Color
  attr_accessor :color
end
```

• A mixin works awkwardly-at-best for ArtistCowboy:
  – Natural for Artist and Cowboy to be Person subclasses
  – Could move methods of one to a mixin, but it is odd style and still doesn't get you two pockets

```ruby
module ArtistM ...

class Artist < Person
  include ArtistM

class ArtistCowboy < Cowboy
  include ArtistM
```