CSE 341, Spring 2008, Lecture 23 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

We have seen that the essence of object-oriented programming is inheritance, overriding, and dynamic dispatch. All our examples have been classes with exactly 1 (immediate) superclass. (In Ruby and Java, classes extend `Object` if they do not declare a different superclass.) But if inheritance is so useful and important, why not allow ways to use more code defined in other places such as another class. In this lecture we consider 3 related ideas:

- *Multiple inheritance:* Languages with multiple inheritance let one class extend multiple other classes. It is the most powerful option, but there are a couple semantic problems that arise that the other ideas avoid. Java and Ruby do not have multiple inheritance; C++ does.
- *Java-style interfaces:* Java classes have one immediate superclass but can "implement" any number of interfaces. Because interfaces do not provide behavior — they only require that certain methods exist — most of the semantic problems go away. Interfaces are fundamentally about type-checking, which we'll study more in the next couple lectures, so there is no reason for them in a language like Ruby. C++ does not have interfaces because inheriting a class with all "abstract" methods (or "pure virtual" methods in C++-speak) accomplishes the same thing.
- *Mixins:* Ruby allows a class to have one immediate superclass but any number of mixins. Because a mixin is "just a pile of methods," many of the semantic problems go away, but it provides behavior unlike Java interfaces. Mixins do not help with all situations where you want multiple inheritance, but they have some excellent uses. In particular, elegant uses of mixins typically involve mixin methods calling methods that they assume are defined in all classes that include the mixin.

Multiple Inheritance:

To understand why multiple inheritance is potentially useful, consider two classic examples (also available in Ruby in the accompanying code file):

- Suppose I have a 2DPoint class with subclasses 3DPoint (adding a z-dimension) and ColorPoint (adding a color field). If I now want a Color3DPoint class, it would seem natural to have two immediate superclasses, 3DPoint and ColorPoint.
- Suppose I have a Person class with subclasses Artist and Cowboy. If I now have an ArtistCowboy (someone who is both), it would seem natural again to have two immediate superclasses. Note, however, that both the Artist class and the Cowboy class have a method "draw" that have very different behaviors (creating a picture versus producing a gun).

If we have multiple inheritance, we have to decide what it means. Naively we might say that the new class has all the fields and methods of all the superclasses. However, if two of the immediate superclasses have the *same* fields or methods, what does that mean? Does it matter if the fields or methods are inherited from the same *common ancestor*? Let us explain these issues in more detail before returning to our examples.

With single inheritance, the *class hierarchy* — all the classes in a program and what extends what — forms a tree, where `A` extends `B` means `A` is a child of `B` in the tree. With multiple inheritance, the class hierarchy becomes a directed acyclic graph (dag). Hence it can have "diamonds" — four classes where one is a (not necessarily immediate) subclass of two others that have a common (not necessarily immediate) superclass. By "immediate" we mean directly extends (child-parent relationship) whereas we could say "transitive" for the more general ancestor-descendant relationship.

Now, with multiple superclasses, we may have conflicts for the fields / methods inherited from the different classes. The draw method for ArtistCowboys is an obvious example where we would like somehow to have both methods in the subclass, or potentially to override one or both of them. At the very least we need

calls using `super` to say which superclass we mean. But this is not necessarily the only conflict. Suppose the Person class has a pocket field that Artists and Cowboys use for different things. Then perhaps an ArtistCowboy should have two pockets, even though the creation of the notion of pocket was in the common ancestor Person.

But if you look at our Color3DPoint example, you would reach the opposite conclusion. Here both 3DPoint and ColorPoint inherit the notion of x and y from a common ancestor, but we certainly don't want a Color3DPoint to have two x methods or two @x fields.

These issues are some of the reasons language with multiple inheritance (most well-known is C++) need quite complicated rules for how subclassing, method lookup, and field access work. For example, C++ has (at least) two different forms of creating a subclass. One always makes copies of all fields from all superclasses. The other makes only one copy of fields that were initially declared by the same common ancestor. Notice this solution would not work well in Ruby, which doesn't have explicit field declarations. There are also options for how to deal with method conflicts, some of which will be relevant when we consider mixins below. We could pick a fixed order (like the leftmost superclass shadows the others) or perhaps require the subclass to override all method names that conflict (or at least manually specify which draw method should dominate, for example).

Java-style interfaces:

In Java, a class can only have one superclass but it can implement any number of interfaces. An interface is just a list of methods and their types. A class type-checks only if it actually provides (directly or via inheritance) all the methods of all the interfaces it claims to implement. An interface is a type, so if a class `C` implements interface `I`, then we can pass an instance of `C` to a method expecting an argument of type `I`, for example. Interfaces are closer to the idea of "duck typing" than just using classes as types, but they still require being named and a class has some interface type only if the class definition explicitly says it implements the interface. We will talk more about OO typing in future lectures.

Because interfaces do not actually *define* methods — they only name them and give them types — none of the problems discussed above about multiple inheritance arise. If two interfaces have a method-name conflict, it doesn't matter — a class can still implement them both. If two interfaces disagree on a method's type, then no class can possibly implement them both but the type-checker will catch that. And in a dynamically typed language, there is really no reason to have interfaces. We can *already* pass any object to any method and call any method on any object. It is up to us to keep track "in our head" (preferably in comments as necessary) what objects can respond to what messages. The essence of dynamic typing is not writing down this stuff.

Bottom line: Implementing interfaces does not inherit code; it is purely something to help more programs type-check in a statically typed language.

Also note that languages with abstract methods and multiple inheritance do not need interfaces. An abstract method (called a pure virtual method in C++) is a method that is promised to exist in all instances of some class but is not defined by the class. Hence any subclass that is actually used to create objects must override this method or inherit from a class that does. (So a class with an abstract method cannot have its constructor used.) Now, if some class has only abstract methods, that is essentially an interface. Conflicts aren't a problem since the method definition does not actually exist — a non-abstract method can trivially "win" over an abstract method. Java has abstract classes and we can define classes with all abstract methods, but Java still needs a separate notion of interfaces because of single inheritance: we want to allow classes to implement multiple interfaces.

Bottom line: C++ doesn't need interfaces because you can get the same effect with abstract classes and multiple inheritance (but recall multiple inheritance has complications that interfaces do not).

Mixins:

Ruby has *mixins*, which are somewhere inbetween multiple inheritance and interfaces. They provide actual code to classes that *include* them, but they are not classes themselves, so they do not have constructors or a separate notion of fields. Ruby did not invent mixins. Its standard-library makes good use of them, though.

To define a Ruby mixin, we use the keyword `module` instead of `class`. (Modules do a bit more than just serve as mixins, hence the strange word choice.) For example, here is a mixin for adding color methods to a class:

```
module Color
  attr_accessor :color
  def darken
    self.color = "dark " + self.color
  end
end
```

This mixin defines three methods, `color`, `color=`, and `darken`. A class definition can include these methods by using the `include` keyword and the name of the mixin. For example:

```
class ColorPt < Pt
  include Color
end
```

This defines a subclass of `Pt` that also has the three methods defined by `Color`. Such classes can have other methods defined/overridden too; here we just chose not to add anything additional. This is not necessarily good style for a couple reasons. First, our `initialize` (inherited from `Pt`) does not create the `@color` field, so we're relying on clients to call `color=` before they call `color` or they will get `nil` back. So overriding `initialize` is probably a good idea. Second, mixins that use fields are a bit questionable stylistically. As you might expect in Ruby, the fields they use will be part of the object the mixin is included in. So if the field name conflicts with some totally separate field defined by the class, the two separate pieces of code will mutate the same data. After all, mixins are "very simple" — they just define a collection of methods that can be included in a class.

Now that we have mixins, we also have to reconsider our method lookup rules. We have to choose something and this is what Ruby chooses: If `obj` is an instance of class `C` and we send message `m` to `obj`,

- First look in the class `C` for a definition of `obj`.
- Next look in mixins included in `C`. Later includes shadow earlier ones.
- Next look in `C`'s superclass.
- Next look in `C`'s superclass' mixins.
- Next look in `C`'s super-superclas.
- Etc.

Many of the elegant uses of mixins do the following strange-sounding thing: They define methods that call other methods on `self` that are *not* defined by the mixin. Instead the mixin *assumes* that all classes that include the mixin define this method. For example, consider this mixin that lets us "double" instances of any class that has `+` defined:

```
module Doubler
  def double
    self + self # uses self's + message, not defined in Doubler
  end
end
```

If we include `Doubler` in some class `C` and call `double` on an instance of the class, we will call the `+` method on the instance, getting an error if it is not defined. But if `+` is defined, everything works out. So now we can easily get the convenience of doubling just by defining `+` and including the `Doubler` mixin. For example:

```
class AnotherPt
  attr_accessor :x, :y
  include Doubler
  def + other  # add two points
    ans = AnotherPt.new
    ans.x = self.x + other.x
    ans.y = self.y + other.y
    ans
  end
end
```

Now instances of `AnotherPt` have `double` methods that do what we want. We could even add `double` to classes that already exists:

```
class String
  include Doubler
end
```

Of course, this example is a little silly since the `double` method is so simple that copying it over and over again wouldn't be disastrous.

The same idea is used a lot in Ruby with two mixins named `Enumerable` and `Comparable`. What `Comparable` does is provide methods `=`, `!=`, `>`, `>=`, `<`, and `<=`, all of which assume the class defines `<=>`. What `<=>` needs to do is return a negative number if its left argument is less than its right, 0 if they are equal, and a positive number if the left argument is greater than the right. So now a class like `Integer` doesn't have to define all these comparisons — it just defines `<=>` and includes `Comparable`. It is very useful for our own classes too, such as this example that can compare names:

```
class Name
  attr_accessor :first, :middle, :last
  include Comparable
  def initialize(first,last,middle="")
    @first = first
    @last = last
    @middle = middle
  end
  def <=> other
    l = @last <=> other.last # <=> defined on strings
    return l if l != 0
    f = @first <=> other.first
    return f if f != 0
    @middle <=> other.middle
  end
end
```

Defining methods in `Comparable` is easy, but we certainly wouldn't want to repeat the work for every class that wants comparisons. For example, the `>` method is just:

```
def > other
  (self <=> other) > 0
end
```

(Okay, there is admittedly a circularity here in the case of Integers, but Integer is actually built into the implementation.)

The `Enumerable` module is where many of the useful block-taking methods that iterate over some data structure are defined. Examples are `any?`, `map`, and `inject`. They are all written assuming the class has the method `each` defined. So a class can define `each`, include the `Enumerable` mixin, and have all these

convenient methods. So the `Array` class for example can just define `each` and include `Enumerable`. Here is another example for a range class we might define:[1]

```
class MyRange
  include Enumerable
  def initialize(low,high)
    @low = low
    @high = high
  end
  def each
    i=@low
    while i <= @high
      yield i
      i=i+1
    end
  end
end
```

Now we can write code like `MyRange.new(4,8).inject {|x,y| x+y}`. Note that the map method in `Enumerable` always returns an instance of `Array`. After all, it doesn't "know how" to produce an instance of any class, but it does know how to produce an array containing one element for everything produced by `each`. We could define it in the `Enumerable` mixin like this:

```
 def map
   arr = []
   each {|x| arr.push x }
   arr
 end
```

Mixins are not as powerful as multiple inheritance because we have to decide upfront what to make a class and what to make a mixin. Given Artist and Cowboy classes, we still have no way to make an ArtistCowboy. And it's unclear which of Artist or Cowboy or both we might be able to make a mixin instead.

[1] We wouldn't actually define this because Ruby lalready has very powerful range classes.