CSE341: Programming Languages

Lecture 23
Multiple Inheritance, Mixins, Interfaces, Abstract Methods

Brett Wortzman
Summer 2019

Slides originally created by Dan Grossman
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

```ruby
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.

```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
```
Mixins

• A *mixin* is (just) a collection of methods
  – Less than a class: no instances of it

• Languages with mixins (e.g., Ruby modules) typically let a class have one superclass but *include* 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 obj's method m, look in obj's class, then mixins that class includes (later includes shadow), then obj'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 CowboyArtist pocket problem (but sometimes unavoidable?)
The two big ones

The two most popular/useful mixins in Ruby:

• Comparable: Defines $<$, $>$, $==$, $!=$, $\geq$, $\leq$ in terms of $\leq>$

• 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 “spend” their “one superclass” for this
– Do not need the complexity of multiple inheritance

• See the code for some examples
Replacement for multiple inheritance?

• A mixin works pretty 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 does not get you two pockets

```ruby
module ArtistM ...
class Artist < Person
  include ArtistM
end
class ArtistCowboy < Cowboy
  include ArtistM
end
```
Statically-Typed OOP

• Now contrast multiple inheritance and mixins with Java/C#-style interfaces

• Important distinction, but interfaces are about static typing, which Ruby does not have

• So will use Java code after quick introduction to static typing for class-based OOP…
  – Sound typing for OOP prevents “method missing” errors
Classes as Types

• In Java/C#/etc. each class is also a type

• Methods have types for arguments and result

```java
class A {
    Object m1(Example e, String s) {...}
    Integer m2(A foo, Boolean b, Integer i) {...}
}
```

• If \( C \) is a (transitive) subclass of \( D \), then \( C \) is a subtype of \( D \)
  – Type-checking allows subtype anywhere supertype allowed
  – So can pass instance of \( C \) to a method expecting instance of \( D \)
Interfaces are (or were) JustTypes

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

• An interface is not a class; it is [er, used to be] only a type
  – Does not contain method definitions, only their signatures (types)
    • Unlike mixins
    • (Changed in Java 8, makes them more like mixins!)
  – Cannot use new on an interface
    • Like mixins
Implementing Interfaces

• A class can explicitly implement any number of interfaces
  – For class to type-check, it must implement every method in
    the interface with the right type
    • More on allowing subtypes later!
  – Multiple interfaces no problem; just implement everything

• If class type-checks, it is a subtype of the interface

```java
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) {...}
}
```
Multiple interfaces

• Interfaces provide no methods or fields
  – So no questions of method/field duplication when implementing multiple interfaces, unlike multiple inheritance

• What interfaces are for:
  – “Caller can give any instance of any class implementing $I$”
    • So callee can call methods in $I$ regardless of class
  – So much more flexible type system

• Interfaces have little use in a dynamically typed language
  – Dynamic typing already much more flexible, with trade-offs we studied
Connections

Let’s now answer these questions:

• What does a statically typed OOP language need to support “required overriding”?

• How is this similar to higher-order functions?

• Why does a language with multiple inheritance (e.g., C++) not need Java/C#-style interfaces?

[Explaining Java’s abstract methods / C++’s pure virtual methods]
Required overriding

Often a class expects all subclasses to override some method(s)

- The purpose of the superclass is to abstract common functionality, but some non-common parts have no default

A Ruby approach:

- Do not define must-override methods in superclass
- Subclasses can add it
- Creating instance of superclass can cause method-missing errors

```ruby
# do not use A.new
# all subclasses should define m2
class A
  def m1 v
    ...
    self.m2 e ...
  end
end
```
Static typing

- In Java/C#/C++, prior approach fails type-checking
  - No method `m2` defined in superclass
  - One solution: provide error-causing implementation

```ruby
class A
  def m1 v
    ... self.m2 e ... 
  end
  def m2 v
    raise "must be overridden"
  end
end
```

- Better: Use static checking to prevent this error…
Abstract methods

- Java/C#/C++ let superclass give signature (type) of method subclasses should provide
  - Called **abstract methods** or **pure virtual methods**
  - Cannot creates instances of classes with such methods
    - Catches error at compile-time
    - Indicates intent to code-reader
    - Does not make language more powerful

```java
abstract class A {
    T1 m1(T2 x) { ... m2(e); ... }
    abstract T3 m2(T4 x);
}
class B extends A {
    T3 m2(T4 x) { ... }
}
```
Passing code to other code

- Abstract methods and dynamic dispatch: An OOP way to have subclass “pass code” to other code in superclass

```java
abstract class A {
    T1 m1(T2 x) { ... m2(e); ... }
    abstract T3 m2(T4 x);
}
class B extends A {
    T3 m2(T4 x) { ... }
}
```

- Higher-order functions: An FP way to have caller “pass code” to callee

```haskell
fun f (g,x) = ... g e ...
fun h x = ... f((fn y => ...),...)
```
No interfaces in C++

• If you have multiple inheritance and abstract methods, you do not also need interfaces

• Replace each interface with a class with all abstract methods

• Replace each “implements interface” with another superclass

So: Expect to see interfaces only in statically typed OOP without multiple inheritance
  – Not Ruby
  – Not C++