```
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

OOP vs. Functional Decomposition

Breaking things down

- In functional (and procedural) programming, break programs down into functions that perform some operation
- In object-oriented programming, break programs down into classes that give behavior to some kind of data

Beginning of this unit:

  - These two forms of *decomposition* are so exactly opposite that they are two ways of looking at the same “matrix”
  - Which form is “better” is somewhat personal taste, but also depends on how you expect to *change/extend software*
  - For some operations over two (multiple) arguments, functions and pattern-matching are straightforward, but with OOP we can do it with *double dispatch* (multiple dispatch)

The expression example

Well-known and compelling example of a common *pattern*:

- Expressions for a small language
- Different variants of expressions: ints, additions, negations, …
- Different operations to perform: **`eval`**, **`toString`**, **`hasZero`**, …

Leads to a matrix (2D-grid) of variants and operations

- Implementation will involve deciding what "should happen" for each entry in the grid *regardless of the PL*

	`eval`	**`toString`**	**`hasZero`**	…
`Int`				
`Add`				
`Negate`				
…				

Standard approach in ML

	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- Define a *datatype*, with one *constructor* for each variant
  - (No need to indicate datatypes if dynamically typed)
- "Fill out the grid" via one function per column
  - Each function has one branch for each column entry
  - Can combine cases (e.g., with wildcard patterns) if multiple entries in column are the same

[See the ML code]

Standard approach in OOP

	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- Define a *class*, with one *abstract method* for each operation
  - (No need to indicate abstract methods if dynamically typed)
- Define a *subclass* for each variant
- So "fill out the grid" via one class per row with one method implementation for each grid position
  - Can use a method in the superclass if there is a default for multiple entries in a column

[See the Ruby code] [*Optional*: See the Java code]

A big course punchline

	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- FP and OOP often doing the same thing in *exact* opposite way
  - Organize the program "by rows" or "by columns"

- Which is "most natural" may depend on what you are doing (e.g., an interpreter vs. a GUI) or personal taste

- Code layout is important, but there is no perfect way since software has many dimensions of structure
  - Tools, IDEs can help with multiple "views" (e.g., rows / columns)

```
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

Adding Operations or Variants

Extensibility

	`eval`	`toString`	`hasZero`	`noNegConstants`
`Int`				
`Add`				
`Negate`				
`Mult`				

- For implementing our grid so far, SML / Racket style usually by column and Ruby / Java style usually by row
- But beyond just style, this decision affects what (unexpected?) software *extensions* need not change old code
- Functions [see ML code]:
  - Easy to add a new operation, e.g., **`noNegConstants`**
  - Adding a new variant, e.g., **`Mult`** requires modifying old functions, but ML type-checker gives a to-do list if original code avoided wildcard patterns

Extensibility

	`eval`	`toString`	`hasZero`	`noNegConstants`
`Int`				
`Add`				
`Negate`				
`Mult`				

- For implementing our grid so far, SML / Racket style usually by column and Ruby / Java style usually by row
- But beyond just style, this decision affects what (unexpected?) software *extensions* are easy and/or do not change old code
- Objects [see Ruby code]:
  - Easy to add a new variant, e.g., **`Mult`**
  - Adding a new operation, e.g., **`noNegConstants`** requires modifying old classes, but [optional:] Java type-checker gives a to-do list if original code avoided default methods

The other way is possible

- Functions allow new operations and objects allow new variants without modifying existing code *even if they didn't plan for it*
  - Natural result of the decomposition

Optional:

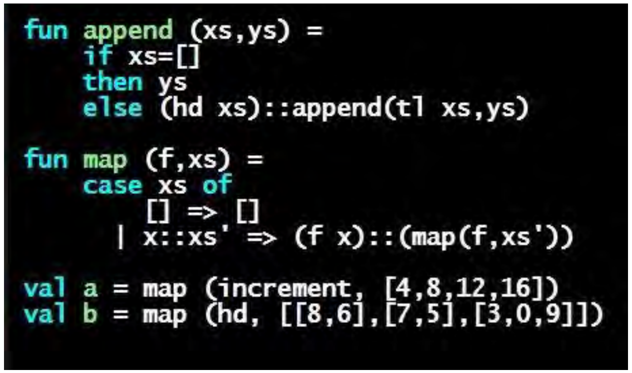
- Functions can support new variants somewhat awkwardly "if they plan ahead"
  - *Not explained here: Can use type constructors to make datatypes extensible and have operations take function arguments to give results for the extensions*

- Objects can support new operations somewhat awkwardly "if they plan ahead"
  - *Not explained here: The popular Visitor Pattern uses the double-dispatch pattern to allow new operations "on the side"*

Thoughts on Extensibility

- Making software extensible is valuable and hard
  - If you know you want new operations, use FP
  - If you know you want new variants, use OOP
  - If both? Languages like Scala try; it’s a hard problem
  - Reality: The future is often hard to predict!

- Extensibility is a double-edged sword
  - Code more reusable without being changed later
  - But makes original code more difficult to reason about locally or change later (could break extensions)
  - Often language mechanisms to make code *less* extensible (ML modules hide datatypes; Java’s **`final`** prevents subclassing/overriding)



Programming Languages

Dan Grossman

Binary Methods with Functional Decomposition

Binary operations

	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- Situation is more complicated if an operation is defined over multiple arguments that can have different variants
  - Can arise in original program or after extension

- Function decomposition deals with this much more simply…

Example

To show the issue:

- Include variants **`String`** and **`Rational`**
- (Re)define **`Add`** to work on any pair of **`Int`**, **`String`**, **`Rational`**
  - Concatenation if either argument a **`String`**, else math

Now just defining the addition operation is a *different* 2D grid:

	`Int`	**`String`**	**`Rational`**
`Int`			
`String`			
`Rational`			

ML Approach

Addition is different for most **`Int`**, **`String`**, **`Rational`** combinations

- Run-time error for non-value expressions

Natural approach: pattern-match on the pair of values

- For *commutative* possibilities, can re-call with **`(v2,v1)`**

```
fun add_values (v1,v2) =
  case (v1,v2) of
     (Int i, Int j) => Int (i+j)
   | (Int i, String s) => String (Int.toString i ^ s)
   | (Int i, Rational(j,k)) => Rational (i*k+j,k)
   | (Rational _, Int _) => add_values (v2,v1)
   | … (* 5 more cases (3*3 total): see the code *)

fun eval e =
  case e of
     …
   | Add(e1,e2) => add_values (eval e1, eval e2)
```

```
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

Binary Methods with OOP: Double Dispatch

Example

To show the issue:

- Include variants **`String`** and **`Rational`**
- (Re)define **`Add`** to work on any pair of **`Int`**, **`String`**, **`Rational`**
  - Concatenation if either argument a **`String`**, else math

Now just defining the addition operation is a *different* 2D grid:

	`Int`	**`String`**	**`Rational`**
`Int`			
`String`			
`Rational`			

Worked just fine with functional decomposition -- what about OOP…

What about OOP?

Starts promising:

– Use OOP to call method **`add_values`** to one value with other value as result

```
class Add
  …
  def eval
    e1.eval.add_values e2.eval
  end
end
```

Classes **`Int`**, **`MyString`**, **`MyRational`** then all implement

– Each handling 3 of the 9 cases: "add **`self`** to argument"

```
class Int
  …
  def add_values v
    … # what goes here?
  end
end
```

First try

- This approach is common, but is "not as OOP"
  - *So do not do it on your homework*

```
class Int
  def add_values v
    if v.is_a? Int
       Int.new(v.i + i)
    elsif v.is_a? MyRational
       MyRational.new(v.i+v.j*i,v.j)
    else
       MyString.new(v.s + i.to_s)
  end
end
```

- A "hybrid" style where we used dynamic dispatch on 1 argument and then switched to Racket-style type tests for other argument
  - Definitely not "full OOP"

Another way…

- **`add_values`** method in **`Int`** needs "what kind of thing" **`v`** has
  - Same problem in **`MyRational`** and **`MyString`**
- In OOP, "always" solve this by calling a method on **`v`** instead!
- But now we need to "tell" **`v`** "what kind of thing" **`self`** is
  - We know that!
  - "Tell" **`v`** by calling different methods on **`v`**, passing **`self`**
- Use a "programming trick" (?) called *double-dispatch*…

Double-dispatch "trick"

- **`Int`**, **`MyString`**, and **`MyRational`** each define all of **`addInt`**, **`addString`**, and **`addRational`**
  - For example, **`String`**'s **`addInt`** is for adding concatenating an integer argument to the string in **`self`**
  - 9 total methods, one for each case of addition
- **`Add`**'s **`eval`** method calls **`e1.eval.add_values e2.eval`**, which dispatches to **`add_values`** in **`Int`**, **`String`**, or **`Rational`**
  - **`Int`**'s **`add_values`**: **`v.addInt self`**
  - **`MyString`**'s **`add_values`**: **`v.addString self`**
  - **`MyRational`**'s **`add_values`**: **`v.addRational self`**

  So **`add_values`** performs "2nd dispatch" to the correct case of 9!

[Definitely see the code]

Why showing you this

- Honestly, partly to belittle full commitment to OOP
- To understand dynamic dispatch via a sophisticated idiom
- Because required for the homework
- To contrast with *multimethods* (optional)

Optional note: Double-dispatch also works fine with static typing

  – See Java code
  – Method declarations with types may help clarify

```
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

Optional: Multimethods

Being Fair

Belittling OOP style for requiring the manual trick of double dispatch is somewhat unfair…

What would work better:

- **`Int`**, **`MyString`**, and **`MyRational`** each define three methods all named **`add_values`**
  - One **`add_values`** takes an **`Int`**, one a **`MyString`**, one a **`MyRational`**
  - So 9 total methods named **`add_values`**
  - **`e1.eval.add_values e2.eval`** picks the right one of the 9 at run-time using the classes of the two arguments
- Such a semantics is called *multimethods* or *multiple dispatch*

Multimethods

General idea:

- Allow multiple methods with same name
- Indicate which ones take instances of which classes
- Use dynamic dispatch on arguments in addition to receiver to pick which method is called

If dynamic dispatch is essence of OOP, this is more OOP

- No need for awkward manual multiple-dispatch

Downside: Interaction with subclassing can produce situations where there is “no clear winner” for which method to call

Ruby: Why not?

Multimethods a bad fit (?) for Ruby because:

- Ruby places no restrictions on what is passed to a method

- Ruby never allows methods with the same name
  - Same name means overriding/replacing

Java/C#/C++: Why not?

- Yes, Java/C#/C++ allow multiple methods with the same name
- No, these language do *not* have multimethods
  - They have *static overloading*
  - Uses static types of arguments to choose the method
    - But of course run-time class of receiver [odd hybrid?]
  - No help in our example, so still code up double-dispatch manually
- Actually, C# 4.0 has a way to get effect of multimethods
- Many other languages have multimethods (e.g., Clojure)
  - They are not a new idea

```
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

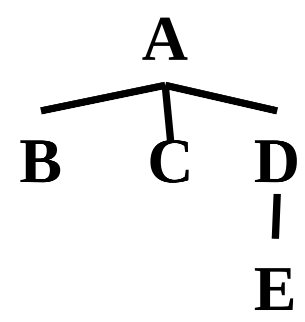
- If inheritance and overriding are so useful, why limit ourselves to one superclass?
  - Because the semantics is often awkward (this topic)
  - Because it makes static type-checking harder (not discussed)
  - Because it makes efficient implementation harder (not discussed)

- 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

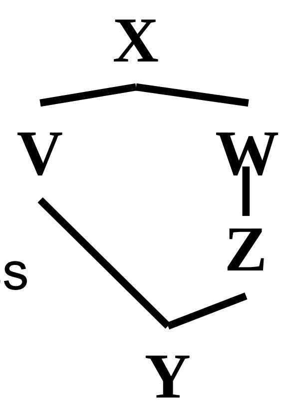
Trees, dags, and diamonds

- Note: The phrases *subclass*, *superclass* can be ambiguous
  - There are *immediate* subclasses, superclasses
  - And there are *transitive* subclasses, superclasses

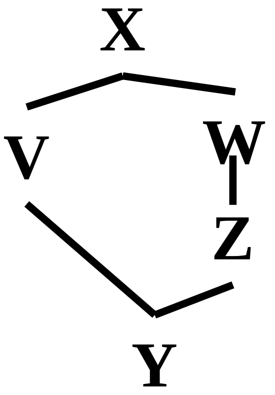
- Single inheritance: the *class hierarchy* is a tree
  - Nodes are classes
  - Parent is immediate superclass
  - Any number of children allowed



- 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
```

```
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

Mixins

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* 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
```

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 `<`, `>`, `==`, `!=`, `>=`, `<=` 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 "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

```
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

```
module ArtistM …
class Artist < Person
  include ArtistM
class ArtistCowboy < Cowboy
  include ArtistM
```

```
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

Interfaces

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 [pseudo]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

```
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 Types

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

- An interface is not a class; it is only a type
  - Does not contain method *definitions*, only their *signatures* (types)
    - Unlike 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

```
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

```
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

Optional: Abstract Methods

Connections

Answered in this segment:

- 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

```
# 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

```
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

```
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

```
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

```
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++