Coursera Programming Languages Course
Section 9 Summary

Standard Description: This summary covers **roughly** *the same material as the lecture videos and the materials (slides, code) posted with each video. It can help to read about the material in a narrative style and to have the material for an entire section of the course in a single document, especially when reviewing the material later. Please report errors in these notes on the discussion board.*

Contents

OOP Versus Functional Decomposition

We can compare procedural (functional) decomposition and object-oriented decomposition using the classic example of implementing operations for a small expression language. In functional programming, we typically break programs down into functions that perform some operation. In OOP, we typically break programs down into classes that give behavior to some kind of data.

We show that the two approaches largely lay out the same ideas in exactly opposite ways, and which way is "better" is either a matter of taste or depends on how software might be changed or *extended* in the future. We then consider how both approaches deal with operations over multiple arguments, which in many object-oriented languages requires a technique called *double (multiple) dispatch* in order to stick with an object-oriented style.

The Basic Set-Up

The following problem is the canonical example of a common programming pattern, and, not coincidentally, is a problem we have already considered a couple times in the course. Suppose we have:

- Expressions for a small "language" such as for arithmetic
- Different *variants* of expressions, such as integer values, negation expressions, and addition expressions
- Different *operations* over expressions, such as evaluating them, converting them to strings, or determining if they contain the constant zero in them

This problem leads to a conceptual *matrix* (two-dimensional grid) with one entry for each combination of variant and operation:

	eval	toString	hasZero
Int			
Add			
Negate			

No matter what programming language you use or how you approach solving this programming problem, you need to indicate what the proper behavior is for each entry in the grid. Certain approaches or languages might make it easier to specify defaults, but you are still deciding something for every entry.

The Functional Approach

In functional languages, the standard style is to do the following:

- Define a *datatype* for expressions, with one *constructor* for each variant. (In a dynamically typed language, we might not give the datatype a name in our program, but we are still thinking in terms of the concept. Similarly, in a language without direct support for constructors, we might use something like lists, but we are still thinking in terms of defining a way to construct each variant of data.)
- Define a *function* for each operation.
- In each function, have a branch (e.g., via pattern-matching) for each variant of data. If there is a default for many variants, we can use something like a wildcard pattern to avoid enumerating all the branches.

Note this approach is really just procedural decomposition: breaking the problem down into procedures corresponding to each operation.

This ML code shows the approach for our example: Notice how we define all the kinds of data in one place and then the nine entries in the table are implemented "by column" with one function for each column:

```
exception BadResult of string

datatype exp =
    Int    of int
  | Negate of exp
  | Add    of exp * exp

fun eval e =
    case e of
        Int _       => e
      | Negate e1   => (case eval e1 of
                          Int i => Int (~i)
                        | _ => raise BadResult "non-int in negation")
      | Add(e1,e2) => (case (eval e1, eval e2) of
                          (Int i, Int j) => Int (i+j)
                        | _ => raise BadResult "non-ints in addition")

fun toString e =
    case e of
        Int i       => Int.toString i
      | Negate e1   => "-(" ^ (toString e1) ^ ")"
      | Add(e1,e2) => "("  ^ (toString e1) ^ " + " ^ (toString e2) ^ ")"
```

datatype for expressions one for each variant

function per operation

branch for each data variant

```
fun hasZero e =
    case e of
        Int i      => i=0
      | Negate e1  => hasZero e1
      | Add(e1,e2) => (hasZero e1) orelse (hasZero e2)
```

The Object-Oriented Approach

In object-oriented languages, the standard style is to do the following:

- Define a *class* for expressions, with one *abstract method* for each operation. (In a dynamically typed language, we might not actually list the abstract methods in our program, but we are still thinking in terms of the concept. Similarly, in a language with duck typing, we might not actually use a superclass, but we are still thinking in terms of defining what operations we need to support.)
- Define a *subclass* for each variant of data.
- In each subclass, have a method definition for each operation. If there is a default for many variants, we can use a method definition in the superclass so that via inheritance we can avoid enumerating all the branches.

Note this approach is a data-oriented decomposition: breaking the problem down into classes corresponding to each data variant.

Here is the Ruby code, which for clarity has the different kinds of expressions subclass the `Exp` class. In a statically typed language, this would be required and the superclass would have to declare the methods that every subclass of `Exp` defines — listing all the operations in one place. Notice how we define the nine entries in the table "by row" with one class for each row.

```
class Exp
  # could put default implementations or helper methods here
end
class Int < Exp
  attr_reader :i
  def initialize i
    @i = i
  end
  def eval
    self
  end
  def toString
    @i.to_s
  end
  def hasZero
    i==0
  end
end
class Negate < Exp
  attr_reader :e
  def initialize e
    @e = e
  end
  def eval
```

```
    Int.new(-e.eval.i) # error if e.eval has no i method (not an Int)
  end
  def toString
    "-(" + e.toString + ")"
  end
  def hasZero
    e.hasZero
  end
end
class Add < Exp
  attr_reader :e1, :e2
  def initialize(e1,e2)
    @e1 = e1
    @e2 = e2
  end
  def eval
    Int.new(e1.eval.i + e2.eval.i) # error if e1.eval or e2.eval have no i method
  end
  def toString
    "(" + e1.toString + " + " + e2.toString + ")"
  end
  def hasZero
    e1.hasZero || e2.hasZero
  end
end
```

The Punch-Line

So we have seen that functional decomposition breaks programs down into functions that perform some operation and object-oriented decomposition breaks programs down into classes that give behavior to some kind of data. These are so exactly opposite that they are the same — just deciding whether to lay out our program "by column" or "by row." Understanding this symmetry is invaluable in conceptualizing software or deciding how to decompose a problem. Moreover, various software tools and IDEs can help you view a program in a different way than the source code is decomposed. For example, a tool for an OOP language that shows you all methods `foo` that override some superclass' `foo` is showing you a column even though the code is organized by rows.

So, which is better? It is often a matter of personal preference whether it seems "more natural" to lay out the concepts by row or by column, so you are entitled to your opinion. What opinion is most common can depend on what the software is about. For our expression problem, the functional approach is probably more popular: it is "more natural" to have the cases for `eval` together rather than the operations for `Negate` together. For problems like implementing graphical user interfaces, the object-oriented approach is probably more popular: it is "more natural" to have the operations for a kind of data (like a `MenuBar`) together (such as `backgroundColor`, `height`, and `doIfMouseIsClicked` rather than have the cases for `doIfMouseIsClicked` together (for `MenuBar`, `TextBox`, `SliderBar`, etc.). The choice can also depend on what programming language you are using, how useful libraries are organized, etc.

Extending the Code With New Operations or Variants

The choice between "rows" and "columns" becomes less subjective if we later extend our program by adding new data variants or new operations.

Consider the functional approach. Adding a new operation is easy: we can implement a new function without editing any existing code. For example, this function creates a new expression that evaluates to the same result as its argument but has no negative constants:

```
fun noNegConstants e =
    case e of
        Int i       => if i < 0 then Negate (Int(~i)) else e
      | Negate e1   => Negate(noNegConstants e1)
      | Add(e1,e2)  => Add(noNegConstants e1, noNegConstants e2)
```

On the other hand, adding a new data variant, such as `Mult of exp * exp` is less pleasant. We need to go back and change all our functions to add a new case. In a statically typed language, we do get some help: after adding the `Mult` constructor, *if* our original code did not use wildcard patterns, then the type-checker will give a non-exhaustive pattern-match warning everywhere we need to add a case for `Mult`.

Again the object-oriented approach is exactly the opposite. Adding a new variant is easy: we can implement a new subclass without editing any existing code. For example, this Ruby class adds multiplication expressions to our language:

```
class Mult < Exp
  attr_reader :e1, :e2
  def initialize(e1,e2)
    @e1 = e1
    @e2 = e2
  end
  def eval
    Int.new(e1.eval.i * e2.eval.i) # error if e1.eval or e2.eval has no i method
  end
  def toString
    "(" + e1.toString + " * " + e2.toString + ")"
  end
  def hasZero
    e1.hasZero || e2.hasZero
  end
end
```

On the other hand, adding a new operation, such as `noNegConstants`, is less pleasant. We need to go back and change all our classes to add a new method. In a statically typed language, we do get some help: after declaring in the `Exp` superclass that all subclasses should have a `noNegConstants` method, the type-checker will give an error for any class that needs to implement the method. (This static typing is using abstract methods and abstract classes, which are discussed later.)

Planning for extensibility

As seen above, functional decomposition allows new operations and object-oriented decomposition allows new variants without modifying existing code and without explicitly planning for it — the programming styles "just work that way." It is possible for functional decomposition to support new variants or object-oriented decomposition to support new operations *if you plan ahead* and use somewhat awkward programming techniques (that seem less awkward over time if you use them often).

We do not consider these techniques in detail here and you are not responsible for learning them. For object-oriented programming, "the visitor pattern" is a common approach. This pattern often is implemented using double dispatch, which is covered for other purposes below. For functional programming, we can define our

datatypes to have an "other" possibility and our operations to take in a function that can process the "other data." Here is the idea in SML:

```
datatype 'a ext_exp =
    Int    of int
  | Negate of 'a ext_exp
  | Add    of 'a ext_exp * 'a ext_exp
  | OtherExtExp  of 'a

fun eval_ext (f,e) = (* notice we pass a function to handle extensions *)
  case e of
    Int i       => i
  | Negate e1   => 0 - (eval_ext (f,e1))
  | Add(e1,e2)  => (eval_ext (f,e1)) + (eval_ext (f,e2))
  | OtherExtExp e => f e
```

With this approach, we could create an extension supporting multiplication by instantiating `'a` with `exp * exp`, passing `eval_ext` the function `(fn (x,y) => eval_ext(f,e1) * eval_ext(f,e2))`, and using `OtherExtExp(e1,e2)` for multiplying `e1` and `e2`. This approach can support different extensions, but it does not support well combining two extensions created separately.

Notice that it does *not* work to wrap the original datatype in a new datatype like this:

```
datatype myexp_wrong =
         OldExp of exp
       | MyMult of myexp_wrong * myexp_wrong
```

This approach does not allow, for example, a subexpression of an `Add` to be a `MyMult`.

Thoughts on Extensibility

It seems clear that if you expect new operations, then a functional approach is more natural and if you expect new data variants, then an object-oriented approach is more natural. The problems are (1) the future is often difficult to predict; we may not know what extensions are likely, and (2) both forms of extension may be likely. Newer languages like Scala aim to support both forms of extension well; we are still gaining practical experience on how well it works as it is a fundamentally difficult issue.

More generally, making software that is both robust and extensible is valuable but difficult. Extensibility can make the original code more work to develop, harder to reason about locally, and harder to change (without breaking extensions). In fact, languages often provide constructs exactly to *prevent* extensibility. ML's modules can hide datatypes, which prevents defining new operations over them outside the module. Java's `final` modifier on a class prevents subclasses.

Binary Methods With Functional Decomposition

The operations we have considered so far used only one value of a type with multiple data variants: `eval`, `toString`, `hasZero`, and `noNegConstants` all operated on one expression. When we have operations that take two (binary) or more (n-ary) variants as arguments, we often have many more cases. With functional decomposition all these cases are still covered together in a function. As seen below, the OOP approach is more cumbersome.

For sake of example, suppose we add string values and rational-number values to our expression language. Further suppose we change the meaning of `Add` expressions to the following:

- If the arguments are ints or rationals, do the appropriate arithmetic.
- If either argument is a string, convert the other argument to a string (unless it already is one) and return the concatenation of the strings.

So it is an error to have a subexpression of `Negate` or `Mult` evaluate to a `String` or `Rational`, but the subexpressions of `Add` can be any kind of value in our language: int, string, or rational.

The interesting change to the SML code is in the `Add` case of `eval`. We now have to consider 9 (i.e., $3 * 3$) subcases, one for each combination of values produced by evaluating the subexpressions. To make this explicit and more like the object-oriented version considered below, we can move these cases out into a helper function `add_values` as follows:

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

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)
      | (String s,  Int i)      => String(s ^ Int.toString i) (* not commutative *)
      | (String s1, String s2)  => String(s1 ^ s2)
      | (String s,  Rational(i,j)) => String(s ^ Int.toString i ^ "/" ^ Int.toString j)
      | (Rational _,    Int _)    => add_values(v2,v1) (* commutative: avoid duplication *)
      | (Rational(i,j), String s) => String(Int.toString i ^ "/" ^ Int.toString j ^ s)
      | (Rational(a,b), Rational(c,d)) => Rational(a*d+b*c,b*d)
      | _ => raise BadResult "non-values passed to add_values"
```

Notice `add_values` is defining all 9 entries in this 2-D grid for how to add values in our language — a different kind of matrix than we considered previously because the rows and columns are variants.

	Int	String	Rational
Int			
String			
Rational			

While the number of cases may be large, that is inherent to the problem. If many cases work the same way, we can use wildcard patterns and/or helper functions to avoid redundancy. One common source of redundancy is *commutativity*, i.e., the order of values not mattering. In the example above, there is only one such case: adding a rational and an int is the same as adding an int and a rational. Notice how we exploit this redundancy by having one case use the other with the call `add_values(v2,v1)`.

Binary Methods in OOP: Double Dispatch

We now turn to supporting the same enhancement of strings, rationals, and enhanced evaluation rules for `Add` in an OOP style. Because Ruby has built-in classes called `String` and `Rational`, we will extend our

code with classes named `MyString` and `MyRational`, but obviously that is not the important point. The first step is to add these classes and have them implement all the existing methods, just like we did when we added `Mult` previously. Then that "just" leaves revising the `eval` method of the `Add` class, which previously assumed the recursive results would be instances of `Int` and therefore have a getter method `i`:

```
def eval
  Int.new(e1.eval.i + e2.eval.i) # error if e1.eval or e2.eval have no i method
end
```

Now we could instead replace this method body with code like our `add_values` helper function in ML, but helper *functions* like this are not OOP style. Instead, we expect `add_values` to be a method in the classes that represent values in our language: An `Int`, `MyRational`, or `MyString` should "know how to add itself to another value." So in `Add`, we write:

```
def eval
  e1.eval.add_values e2.eval
end
```

This is a good start and now obligates us to have `add_values` methods in the classes `Int`, `MyRational`, and `MyString`. By putting `add_values` methods in the `Int`, `MyString`, and `MyRational` classes, we nicely divide our work into three pieces using dynamic dispatch depending on the class of the object that `e1.eval` returns, i.e., the receiver of the `add_values` call in the `eval` method in `Add`. But then each of these three needs to handle three of the nine cases, based on the class of the second argument. One approach would be to, in these methods, abandon object-oriented style (!) and use run-time tests of the classes to include the three cases. The Ruby code would look like this:

```
class Int
   ...
   def add_values v
      if v.is_a? Int
          ...
      elsif v.is_a? MyRational
          ...
      else
          ...
      end
   end
end
class MyRational
   ...
   def add_values v
      if v.is_a? Int
          ...
      elsif v.is_a? MyRational
          ...
      else
          ...
      end
   end
end
class MyString
```

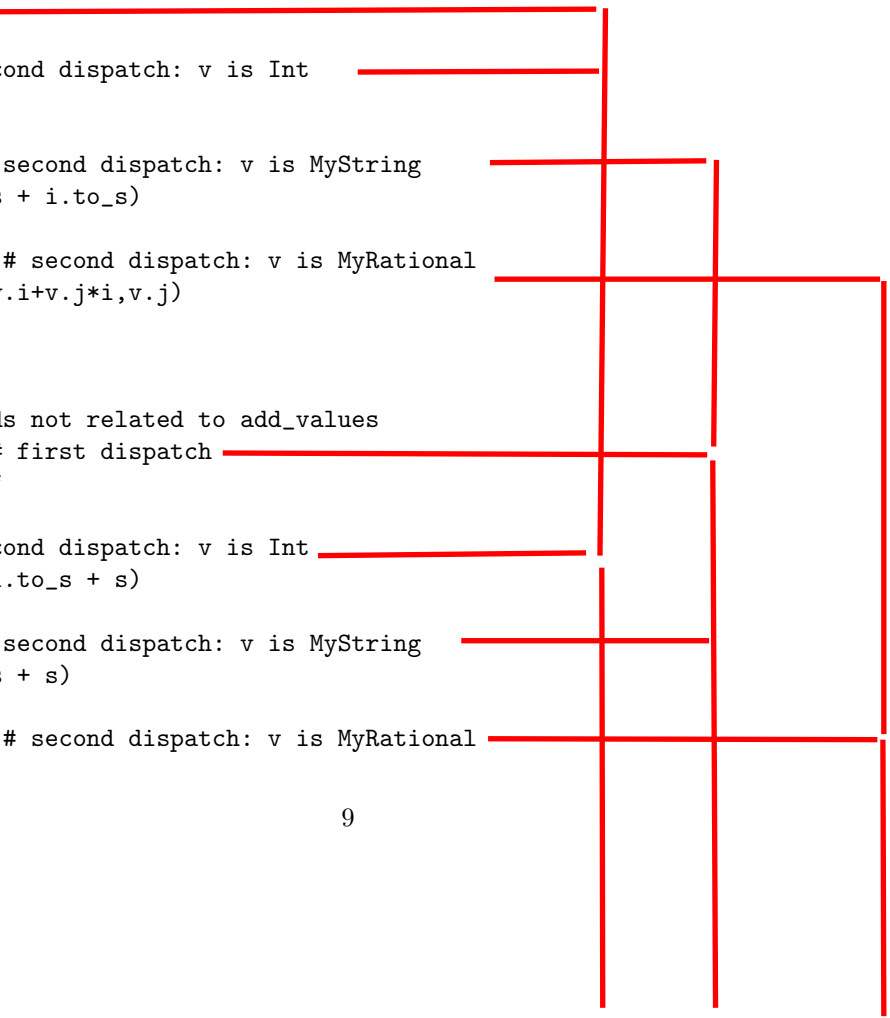
```
   ...
   def add_values v
      if v.is_a? Int
          ...
      elsif v.is_a? MyRational
          ...
      else
          ...
      end
   end
end
```

While this approach works, it is really not object-oriented programming. Rather, it is a mix of object-oriented decomposition (dynamic dispatch on the first argument) and functional decomposition (using `is_a?` to figure out the cases in each method). There is not necessarily anything wrong with that — it is probably simpler to understand than what we are about to demonstrate — but it does give up the extensibility advantages of OOP and really is not "full" OOP.

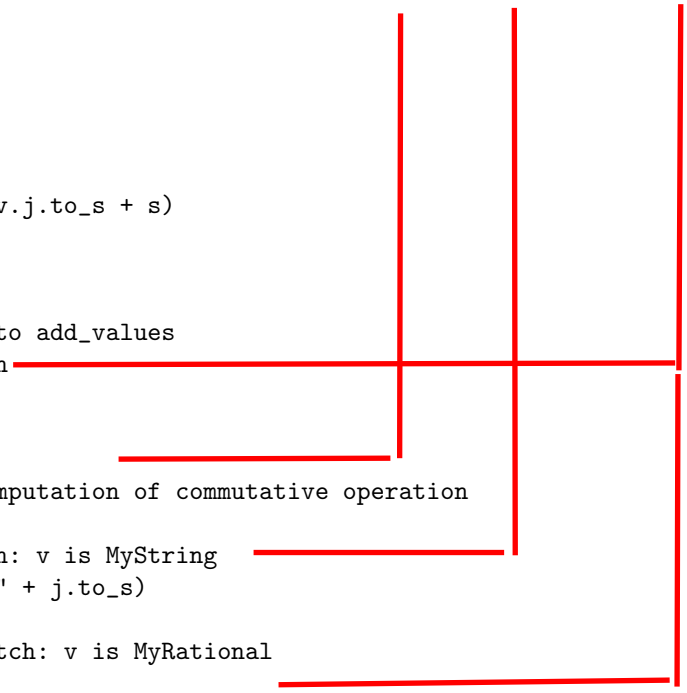
Here is how to think about a "full" OOP approach: The problem inside our three `add_values` methods is that we need to "know" the class of the argument `v`. In OOP, the strategy is to replace "needing to know the class" with calling a method on `v` instead. So we should "tell `v`" to do the addition, passing `self`. And we can "tell `v`" what class `self` is because the `add_values` methods know that: In `Int`, `self` is an `Int` for example. And the way we "tell `v` the class" is to call different methods on `v` for each kind of argument.

This technique is called *double dispatch.* Here is the code for our example, followed by additional explanation:

```
class Int
  ... # other methods not related to add_values
  def add_values v # first dispatch
    v.addInt self
  end
  def addInt v # second dispatch: v is Int
    Int.new(v.i + i)
  end
  def addString v # second dispatch: v is MyString
    MyString.new(v.s + i.to_s)
  end
  def addRational v # second dispatch: v is MyRational
    MyRational.new(v.i+v.j*i,v.j)
  end
end
class MyString
  ... # other methods not related to add_values
  def add_values v # first dispatch
    v.addString self
  end
  def addInt v # second dispatch: v is Int
    MyString.new(v.i.to_s + s)
  end
  def addString v # second dispatch: v is MyString
    MyString.new(v.s + s)
  end
  def addRational v # second dispatch: v is MyRational
```



```
    MyString.new(v.i.to_s + "/" + v.j.to_s + s)
  end
end
class MyRational
  ... # other methods not related to add_values
  def add_values v # first dispatch
    v.addRational self
  end
  def addInt v # second dispatch
    v.addRational self  # reuse computation of commutative operation
  end
  def addString v # second dispatch: v is MyString
    MyString.new(v.s + i.to_s + "/" + j.to_s)
  end
  def addRational v # second dispatch: v is MyRational
    a,b,c,d = i,j,v.i,v.j
    MyRational.new(a*d+b*c,b*d)
  end
end
```



Before understanding how all the method calls work, notice that we do now have our 9 cases for addition in 9 different methods:

- The `addInt` method in `Int` is for when the left operand to addition is an `Int` (in `v`) and the right operation is an `Int` (in `self`).
- The `addString` method in `Int` is for when the left operand to addition is a `MyString` (in `v`) and the right operation is an `Int` (in `self`).
- The `addRational` method in `Int` is for when the left operand to addition is a `MyRational` (in `v`) and the right operation is an `Int` (in `self`).
- The `addInt` method in `MyString` is for when the left operand to addition is an `Int` (in `v`) and the right operation is a `MyString` (in `self`).
- The `addString` method in `MyString` is for when the left operand to addition is a `MyString` (in `v`) and the right operation is a `MyString` (in `self`).
- The `addRational` method in `MyString` is for when the left operand to addition is a `MyRational` (in `v`) and the right operation is a `MyString` (in `self`).
- The `addInt` method in `MyRational` is for when the left operand to addition is an `Int` (in `v`) and the right operation is a `MyRational` (in `self`).
- The `addString` method in `MyRational` is for when the left operand to addition is a `MyString` (in `v`) and the right operation is a `MyRational` (in `self`).
- The `addRational` method in `MyRational` is for when the left operand to addition is a `MyRational` (in `v`) and the right operation is a `MyRational` (in `self`).

As we might expect in OOP, our 9 cases are "spread out" compared to in the ML code. Now we need to understand how dynamic dispatch is picking the correct code in all 9 cases. Starting with the `eval` method in `Add`, we have `e1.eval.add_values e2.eval`. There are 3 `add_values` methods and dynamic dispatch will pick one based on the class of the value returned by `e1.eval`. This the "first dispatch." Suppose `e1.eval` is

an `Int`. Then the next call will be `v.addInt self` where `self` is `e1.eval` and `v` is `e2.eval`. Thanks again to dynamic dispatch, the method looked up for `addInt` will be the right case of the 9. This is the "second dispatch." All the other cases work analogously.

Understanding double dispatch can be a mind-bending exercise that re-enforces how dynamic dispatch works, the key thing that separates OOP from other programming. It is not necessarily intuitive, but it what one must do in Ruby/Java to support binary operations like our addition in an OOP style.

Optional notes:

- OOP languages with *multimethods*, discussed optionally next, do not require the manual double dispatch we have seen here.
- Statically typed languages like Java do not get in the way of the double-dispatch idiom. In fact, needing to declare method argument and return types as well as indicating in the superclass the methods that all subclasses implement can make it easier to understand what is going on. A full Java implementation of our example is posted with the course materials. (It is common in Java to reuse method names for different methods that take arguments of different types. Hence we could use `add` instead of `addInt`, `addString`, and `addRational`, but this can be more confusing than helpful when first learning double dispatch.)

Optional: Multimethods

It is *not* true that all OOP languages require the cumbersome double-dispatch pattern to implement binary operations in a full OOP style. Languages with *multimethods*, also known as *multiple dispatch*, provide more intuitive solutions. In such languages, the classes `Int`, `MyString`, and `MyRational` could each define three methods all named `add_values` (so there would be nine total methods in the program named `add_values`). Each `add_values` method would indicate the class it expects for its argument. Then `e1.eval.add_values e2.eval` would pick the right one of the 9 by, at run-time, considering the class of the result of `e1.eval` *and* the class of the result of `e2.eval`.

This is a powerful and *different* semantics than we have studied for OOP. In our study of Ruby (and Java/C#/C++ work the same way), the method-lookup rules involve the run-time class of the receiver (the object whose method we are calling), not the run-time class of the argument(s). Multiple dispatch is "even more dynamic dispatch" by considering the class of multiple objects and using all that information to choose what method to call.

Ruby does not support multimethods because Ruby is committed to having only one method with a particular name in each class. Any object can be passed to this one method. So there is no way to have 3 `add_values` methods in the same class and no way to indicate which method should be used based on the argument.

Java and C++ also do not have multimethods. In these languages you *can* have multiple methods in a class with the same name and the method-call semantics does use the types of the arguments to choose what method to call. But it uses the *types* of the arguments, which are determined at *compile-time* and *not* the run-time class of the result of evaluating the arguments. This semantics is called *static overloading*. It is considered useful and convenient, but it is not multimethods and does not avoid needing double dispatch in our example.

C# has the same static overloading as Java and C++, but as of version 4.0 of the language one can achieve the effect of multimethods by using the type "dynamic" in the right places. We do not discuss the details here, but it is a nice example of combining language features to achieve a useful end.

Many OOP languages have had multimethods for many years — they are not a new idea. Perhaps the most well-known modern language with multimethods is Clojure.

Multiple Inheritance

We have seen that the essence of object-oriented programming is inheritance, overriding, and dynamic dispatch. All our examples have been classes with 1 (immediate) 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. We now begin discussing 3 related but distinct ideas:

- *Multiple inheritance:* Languages with multiple inheritance let one class extend multiple other classes. It is the most powerful option, but there are some semantic problems that arise that the other ideas avoid. Java and Ruby do not have multiple inheritance; C++ does.
- *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. 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. Ruby's standard libraries make good use of this technique and your code can too.
- *Java/C#-style interfaces:* Java/C# 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 will study more later in this section, so there very little 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 as described more below.

To understand why multiple inheritance is potentially useful, consider two classic examples:

- Consider a `Point2D` class with subclasses `Point3D` (adding a z-dimension) and `ColorPoint` (adding a `color` attribute). To create a `ColorPoint3D` class, it would seem natural to have two immediate superclasses, `Point3D` and `ColorPoint` so we inherit from both.
- Consider a `Person` class with subclasses `Artist` and `Cowboy`. To create 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).

Without multiple inheritance, you end up copying code in these examples. For example, `ColorPoint3D` can subclass `Point3D` and copy code from `ColorPoint` or it can subclass `ColorPoint` and copy code from `Point3D`.

If we have multiple inheritance, we have to decide what it means. Naively we might say that the new class has all the methods of all the superclasses (and fields too in languages where fields are part of class definitions). 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 may not be a tree. 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.

With multiple superclasses, we may have conflicts for the fields / methods inherited from the different classes. The `draw` method for `ArtistCowboy` objects 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 expressions using `super` to indicate which superclass is intended. 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 `ColorPoint3D` example, you would reach the opposite conclusion. Here both `Point3D` and `ColorPoint` inherit the notion of x and y from a common ancestor, but we certainly do not want a `ColorPoint3D` 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 fairly 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. (This solution would not work well in Ruby because instance variables are not part of class definitions.)

Mixins

pile of methods

Ruby has *mixins*, which are somewhere between multiple inheritance (see above) and interfaces (see below). They provide actual code to classes that *include* them, but they are not classes themselves, so you cannot create instances of them. Ruby did not invent mixins. Its standard-library makes good use of them, though. A near-synonym for mixins is *traits*, but we will stick with what Ruby calls mixins.

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 are 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 instance variables are stylistically questionable. As you might expect in Ruby, the instance variables they use will be part of the object the mixin is included in. So if there is a name conflict with some intended-to-be separate instance variable defined by the class

(or another mixin), 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 `m`.
- 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-superclass.
- 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 exist:

```
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 would not be so burdensome.

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 does not have to define all these comparisons — it just defines `<=>` and includes `Comparable`. Consider this example for comparing 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 would not want to repeat the work for every class that wants comparisons. For example, the `>` method is just:

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

The `Enumerable` module is where many of the useful block-taking methods that iterate over some data structure are defined. Examples are `any?`, `map`, `count`, 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
```

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

Now we can write code like `MyRange.new(4,8).inject {|x,y| x+y}` or `MyRange.new(5,12).count {|i| i.odd?}`. Note that the `map` method in `Enumerable` always returns an instance of `Array`. After all, it does not "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 natural way to make an `ArtistCowboy`. And it is unclear which of `Artist` or `Cowboy` or both we might want to define in terms of a mixin.

Java/C#-Style Interfaces

In Java or C#, a class can have only one immediate superclass but it can implement any number of *interfaces*. An interface is just a list of methods and each method's argument types and return type. 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 (in Java and C# every class is also a type), but a class has some interface type only if the class definition *explicitly* says it implements the interface. We discuss more about OOP type-checking later in this section.

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 does not 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. Because interfaces do not define methods, they cannot be used like mixins.

In a dynamically typed language, there is really little reason to have interfaces.[2] 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 related to type-checking in statically typed languages like Java and C#. It makes the type systems in these languages more flexible. So Ruby does not need interfaces.

Optional: Abstract Methods

Often a class definition has methods that call other methods that are not actually defined in the class. It would be an error to create instances of such a class and use the methods such that "method missing" errors occur. So why define such a class? Because the entire point of the class is to be subclassed and have different

[2] Probably the only use would be to change the meaning of Ruby's `is_a?` to incorporate interfaces, but we can more directly just use reflection to find out an object's methods.

subclasses define the missing methods in different ways, relying on dynamic dispatch for the code in the superclass to call the code in the subclass. This much works just fine in Ruby — you can have comments indicating that certain classes are there only for the purpose of subclassing.

The situation is more interesting in statically typed languages. In these languages, the purpose of type-checking is to prevent "method missing" errors, so when using this technique we need to indicate that instances of the superclass must not be created. In Java/C# such classes are called "abstract classes." We also need to give the type of any methods that (non-abstract) subclasses must provide. These are "abstract methods." Thanks to subtyping in these languages, we can have expressions with the *type* of the superclass and know that at run-time the object will actually be one of the subclasses. Furthermore, type-checking ensures the object's class has implemented all the abstract methods, so it is safe to call these methods. In C++, abstract methods are called "pure virtual methods" and serve much the same purpose.

There is an interesting parallel between abstract methods and higher-order functions. In both cases, the language supports a programming pattern where some code is passed other code in a flexible and reusable way. In OOP, different subclasses can implement an abstract method in different ways and code in the superclass, via dynamic dispatch, can then uses these different implementations. With higher-order functions, if a function takes another function as an argument, different callers can provide different implementations that are then used in the function body.

Languages with abstract methods and multiple inheritance (e.g., C++) do not need interfaces. Instead we can just use classes that have nothing but abstract (pure virtual) methods in them like they are interfaces and have classes implementing these "interfaces" just subclass the classes. This subclassing is not inheriting any code exactly because abstract methods do not define methods. With multiple inheritance, we are not "wasting" our one superclass with this pattern.