CSE341: Programming Languages Winter 2017
Unit 8 Summary

Standard Description: This summary covers **roughly** *the same material as class and recitation section. It can help to read about the material in a narrative style and to have the material for an entire unit of the course in a single document, especially when reviewing the material later. Please report errors in these notes, even typos. This summary is not a sufficient substitute for attending class, reading the associated code, etc.*

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) ^ ")"

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.

Notes:

- OOP languages with *multimethods*, discussed 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.)

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 unit, 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

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
```

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
```

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

```
    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 unit.

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.

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 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

[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.

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.

Introduction to Subtyping

We previously studied static types for functional programs, in particular ML's type system. ML uses its type system to prevent errors like treating a number as a function. A key source of expressiveness in ML's type system (not rejecting too many programs that do nothing wrong and programmers are likely to write) is *parametric polymorphism*, also known as *generics*.

So we should also study static types for object-oriented programs, such as those found in Java. If everything is an object (which is less true in Java than in Ruby), then the main thing we would want our type system to prevent is "method missing" errors, i.e., sending a message to an object that has no method for that message. If objects have fields accessible from outside the object (e.g., in Java), then we also want to prevent "field missing" errors. There are other possible errors as well, like calling a method with the wrong number of arguments.

While languages like Java and C# have generics these days, the source of type-system expressiveness most fundamental to object-oriented style is *subtype polymorphism*, also known as *subtyping*. ML does not have subtyping, though this decision is really one of language design (it would complicate type inference, for example).

It would be natural to study subtyping using Java since it is a well-known object-oriented language with a type system that has subtyping. But it is also fairly complicated, using classes and interfaces for types that describe objects with methods, overriding, static overloading, etc. While these features have pluses and minuses, they can complicate the fundamental ideas that underlie how subtyping should work in any language.

So while we will briefly discuss subtyping in OOP, we will mostly use a small language with *records* (like in ML, things with named fields holding contents — basically objects with public fields, no methods, and no class names) and functions (like in ML or Racket). This will let us see how subtyping should — and should not — work.

This approach has the disadvantage that we cannot use any of the language we have studied: ML does not have subtyping and record fields are immutable, Racket and Ruby are dynamically typed, and Java is too complicated for our starting point. So we are going to make up a language with just records, functions, variables, numbers, strings, etc. and explain the meaning of expressions and types as we go.

A Made-Up Language of Records

To study the basic ideas behind subtyping, we will use records with mutable fields, as well as functions and other expressions. Our syntax will be a mix of ML and Java that keeps examples short and, hopefully, clear. For records, we will have expressions for making records, getting a field, and setting a field as follows:

- In the expression `{f1=e1, f2=e2, ..., fn=en}`, each `fi` is a field name and each `ei` is an expression. The semantics is to evaluate each `ei` to a value `vi` and the result is the record value `{f1=v1, f2=v2, ..., fn=vn}`. So a record value is just a collection of fields, where each field has a name and a contents.
- For the expression `e.f`, we evaluate `e` to a value `v`. If `v` is a record with an `f` field, then the result is the contents of the `f` field. Our type system will ensure `v` has an `f` field.
- For the expression `e1.f = e2`, we evaluate `e1` and `e2` to values `v1` and `v2`. If `v1` is a record with an `f` field, then we update the `f` field to have `v2` for its contents. Our type system will ensure `v1` has an `f` field. Like in Java, we will choose to have the result of `e1.f = e2` be `v2`, though usually we do not use the result of a field-update.

Now we need a type system, with a form of types for records and typing rules for each of our expressions. Like in ML, let's write record types as `{f1:t1, f2:t2, ..., fn:tn}`. For example, `{x : real, y : real}` would describe records with two fields named `x` and `y` that hold contents of type `real`. And `{foo: {x : real, y : real}, bar : string, baz : string}` would describe a record with three fields where the `foo` field holds a (nested) record of type `{x : real, y : real}`. We then type-check expressions as follows:

- If `e1` has type `t1`, `e2` has type `t2`, ..., `en` has type `tn`, then `{f1=e1, f2=e2, ..., fn=en}` has type `{f1:t1, f2:t2, ..., fn:tn}`.
- If `e` has a record type containing `f : t`, then `e.f` has type `t` (else `e.f` does not type-check).
- If `e1` has a record type containing `f : t` and `e2` has type `t`, then `e1.f = e2` has type `t` (else `e1.f = e2` does not type-check).

Assuming the "regular" typing rules for other expressions like variables, functions, arithmetic, and function calls, an example like this will type-check as we would expect:

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val pythag : {x:real,y:real} = {x=3.0, y=4.0}
val five : real = distToOrigin(pythag)
```

In particular, the function `distToOrigin` has type `{x : real, y : real} -> real`, where we write function types with the same syntax as in ML. The call `distToOrigin(pythag)` passes an argument of the right type, so the call type-checks and the result of the call expression is the return type `real`.

This type system does what it is intended to do: No program that type-checks would, when evaluated, try to look up a field in a record that does not have that field.

Wanting Subtyping

With our typing rules so far, this program would not type-check:

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val c : {x:real,y:real,color:string} = {x=3.0, y=4.0, color="green"}
val five : real = distToOrigin(c)
```

In the call `distToOrigin(c)`, the type of the argument is `{x:real,y:real,color:string}` and the type the function expects is `{x:real,y:real}`, breaking the typing rule that functions must be called with the type of argument they expect. Yet the program above is safe: running it would not lead to accessing a field that does not exist.

A natural idea is to make our type system more lenient as follows: If some expression has a record type `{f1:t1, ..., fn:tn}`, then let the expression *also* have a type where some of the fields are removed. Then our example will type-check: Since the expression `c` has type `{x:real,y:real,color:string}`, it can also have type `{x:real,y:real}`, which allows the call to type-check. Notice we could also use `c` as an argument to a function of type `{color:string}->int`, for example.

Letting an expression that has one type also have another type that has less information is the idea of *subtyping*. (It may seem backwards that the *subtype* has *more* information, but that is how it works. A less-backwards way of thinking about it is that there are "*fewer*" values of the subtype than of the supertype because values of the subtype have more obligations, e.g., having more fields.)

The Subtyping Relation

We will now add subtyping to our made-up language, in a way that will not require us to change any of our existing typing rules. For example, we will leave the function-call rule the same, still requiring that the type of the actual argument *equal* the type of the function parameter in the function definition. To do this, we will add two things to our type system:

- The idea of one type being a subtype of another: We will write `t1 <: t2` to mean `t1` is a subtype of `t2`.
- One and only new typing rule: If `e` has type `t1` and `t1 <: t2`, then `e` (also) has type `t2`.

So now we just need to give rules for `t1 <: t2`, i.e., when is one type a subtype of another. This approach is good language engineering — we have separated the idea of subtyping into a single binary relation that we can define separately from the rest of the type system.

A common misconception is that if we are defining our own language, then we can make the typing and subtyping rules whatever we want. That is only true if we forget that our type system is allegedly preventing something from happening when programs run. If our goal is (still) to prevent field-missing errors, then we cannot add any subtyping rules that would cause us to stop meeting our goal. This is what people mean when they say, "Subtyping is not a matter of opinion."

For subtyping, the key guiding principle is *substitutability*: If we allow `t1 <: t2`, then any value of type `t1` must be able to be used in every way a `t2` can be. For records, that means `t1` should have all the fields that `t2` has and with the same types.

Some Good Subtyping Rules

Without further ado, we can now give four subtyping rules that we can add to our language to accept more programs without breaking the type system. The first two are specific to records and the next two, while perhaps seeming unnecessary, do no harm and are common in any language with subtyping because they combine well with other rules:

- "Width" subtyping: A supertype can have a subset of fields with the same types, i.e., a subtype can have "extra" fields
- "Permutation" subtyping: A supertype can have the same set of fields with the same types in a different order.
- Transitivity: If `t1 <: t2` and `t2 <: t3`, then `t1 <: t3`.
- Reflexivity: Every type is a subtype of itself: `t <: t`.

Notice that width subtyping lets us forget fields, permutation subtyping lets us reorder fields (e.g., so we can pass a `{x:real,y:real}` in place of a `{y:real,x:real}`) and transitivity with those rules lets us do both (e.g., so we can pass a `{x:real,foo:string,y:real}` in place of a `{y:real,x:real}`).

Depth Subtyping: A Bad Idea With Mutation

Our subtyping rules so far let us drop fields or reorder them, but there is no way for a supertype to have a field with a different type than in the subtype. For example, consider this example, which passes a "sphere" to a function expecting a "circle." Notice that circles and spheres have a `center` field that itself holds a record.

```
fun circleY (c:{center:{x:real,y:real}, r:real}) =
   c.center.y

val sphere:{center:{x:real,y:real,z:real}, r:real}) = {center={x=3.0,y=4.0,z=0.0}, r=1.0}
val _ = circleY(sphere)
```

The type of `circleY` is `{center:{x:real,y:real}, r:real}->real` and the type of `sphere` is `{center:{x:real,y:real,z:real}, r:real}`, so the call `circleY(sphere)` can type-check only if

```
{center:{x:real,y:real,z:real}, r:real} <: {center:{x:real,y:real}, r:real}
```

This subtyping does not hold with our rules so far: We can drop the `center` field, drop the `r` field, or reorder those fields, but we cannot "reach into a field type to do subtyping."

Since we might like the program above to type-check since evaluating it does nothing wrong, perhaps we should add another subtyping rule to handle this situation. The natural rule is "depth" subtyping for records:

- "Depth" subtyping: If `ta <: tb`, then `{f1:t1,...,f:ta,...,fn:tn} <: {f1:t1,...,f:tb,...,fn:tn}`.

This rule lets us use width subtyping on the field `center` to show

```
{center:{x:real,y:real,z:real}, r:real} <: {center:{x:real,y:real}, r:real}
```

so the program above now type-checks.

Unfortunately, this rule breaks our type system, allowing programs that we do not want to allow to type-check! This may not be intuitive and programmers make this sort of mistake often — thinking depth subtyping should be allowed. Here is an example:

```
fun setToOrigin (c:{center:{x:real,y:real}, r:real})=
   c.center = {x=0.0, y=0.0}

val sphere:{center:{x:real,y:real,z:real}, r:real}) = {center={x=3.0,y=4.0,z=0.0}, r=1.0}
val _ = setToOrigin(sphere)
val _ = sphere.center.z
```

This program type-checks in much the same way: The call `setToOrigin(sphere)` has an argument of type `{center:{x:real,y:real,z:real}, r:real}` and uses it as a `{center:{x:real,y:real}, r:real}`. But what happens when we run this program? `setToOrigin` mutates its argument so the `center` field holds a record *with no z field!* So the last line, `sphere.center.z` will not work: it tries to read a field that does not exist.

The moral of the story is simple if often forgotten: In a language with records (or objects) with getters and setters for fields, depth subtyping is unsound — you cannot have a different type for a field in the subtype and the supertype.

Note, however, that if a field is not settable (i.e., it is immutable), then the depth subtyping rule is sound and, like we saw with `circleY`, useful. So this is yet another example of how not having mutation makes programming easier. In this case, it allows more subtyping, which lets us reuse code more.

Another way to look at the issue is that given the three features of (1) setting a field, (2) letting depth subtyping change the type of a field, and (3) having a type system actually prevent field-missing errors, you can have any two of the three.

The Problem With Java/C# Array Subtyping

Now that we understand depth subtyping is unsound if record fields are mutable, we can question how Java and C# treat subtyping for arrays. For the purpose of subtyping, arrays are very much like records, just with field names that are numbers and all fields having the same type. (Since `e1[e2]` computes what index to access and the type system does not restrict what index might be the result, we need all fields to have the same type so that the type system knows the type of the result.) So it should very much surprise us that this code type-checks in Java:

```
class Point { ... } // has fields double x, y
class ColorPoint extends Point { ... } // adds field String color
...
void m1(Point[] pt_arr) {
  pt_arr[0] = new Point(3,4);
}
String m2(int x) {
  ColorPoint[] cpt_arr = new ColorPoint[x];
  for(int i=0; i < x; i++)
     cpt_arr[i] = new ColorPoint(0,0,"green");
  m1(cpt_arr);
  return cpt_arr[0].color;
}
```

The call `m1(cpt_arr)` uses subtyping with `ColorPoint[] <: Point[]`, which is essentially depth subtyping even though array indices are mutable. As a result, it appears that `cpt_arr[0].color` will read the `color` field of an object that does not have such a field.

What actually happens in Java and C# is the assignment `pt_arr[0] = new Point(3,4);` will raise an exception if `pt_arr` is actually an array of `ColorPoint`. In Java, this is an `ArrayStoreException`. The advantage of having the store raise an exception is that no other expressions, such as array reads or object-field reads, need run-time checks. The invariant is that an object of type `ColorPoint[]` always holds objects that have type `ColorPoint` or a subtype, not a supertype like `Point`. Since Java allows depth subtyping on arrays, it cannot maintain this invariant statically. Instead, it has a run-time check on all array assignments, using the "actual" type of array elements and the "actual" class of the value being assigned. So even though in the type system `pt_arr[0]` and `new Point(3,4)` both have type `Point`, this assignment can fail at run-time.

As usual, having run-time checks means the type system is preventing fewer errors, requiring more care and testing, plus the run-time cost of performing these checks on array updates. So why were Java and C# designed this way? It seemed important for flexibility before these languages had generics so that, for example, if you wrote a method to sort an array of `Point` objects, you could use your method to sort an array of `ColorPoint` objects. Allowing this makes the type system simpler and less "in your way" at the expense of statically checking less. Better solutions would be to use generics in combination with subtyping (see bounded polymorphism in the next lecture) or to have support for indicating that a method will not update array elements, in which case depth subtyping is sound.

`null` in Java/C#

While we are on the subject of pointing out places where Java/C# choose dynamic checking over the "natural" typing rules, the far more ubiquitous issue is how the constant `null` is handled. Since this value has no fields or methods (in fact, unlike `nil` in Ruby, it is not even an object), its type should naturally reflect that it cannot be used as the receiver for a method or for getting/setting a field. Instead, Java and C# allow `null` to have *any* object type, as though it defines *every* method and has *every* field. From a static checking perspective, this is exactly backwards. As a result, the language definition has to indicate that *every* field access and method call includes a run-time check for `null`, leading to the `NullPointerException` errors that Java programmers regularly encounter.

So why were Java and C# designed this way? Because there are situations where it is very convenient to have `null`, such as initializing a field of type `Foo` before you can create a `Foo` instance (e.g., if you are building a cyclic list). But it is also very common to have fields and variables that should never hold `null` and you would like to have help from the type-checker to maintain this invariant. Many proposals for incorporating "cannot be `null`" types into programming languages have been made, but none have yet "caught on" for Java or C#. In contrast, notice how ML uses option types for similar purposes: The types `t option` and `t` are not the same type; you have to use `NONE` and `SOME` constructors to build a datatype where values might or might not actually have a `t` value.

Function Subtyping

The rules for when one function type is a subtype of another function type are even less intuitive than the issue of depth subtyping for records, but they are just as important for understanding how to safely override methods in object-oriented languages (see below).

When we talk about function subtyping, we are talking about using a function of one type in place of a function of another type. For example, if `f` takes a function `g` of type `t1->t2`, can we pass a function of type `t3->t4` instead? If `t3->t4` is a subtype of `t1->t2` then this is allowed because, as usual, we can pass the function `f` an argument that is a subtype of the type expected. But this is not "function subtyping" on `f` — it is "regular" subtyping on function arguments. The "function subtyping" is deciding that one function type is a subtype of another.

To understand function subtyping, let's use this example of a higher-order function, which computes the distance between the two-dimensional point `p` and the result of calling `f` with `p`:

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end
```

The type of `distMoved` is

```
(({x:real,y:real}->{x:real,y:real}) * {x:real,y:real}) -> real
```

So a call to `distMoved` requiring no subtyping could look like this:

```
fun flip p = {x=~p.x, y=~p.y}
val d = distMoved(flip, {x=3.0, y=4.0})
```

The call above could also pass in a record with extra fields, such as `{x=3.0,y=4.0,color="green"}`, but this is just ordinary width subtyping on the second argument to `distMoved`. Our interest here is deciding what functions with types other than `{x:real,y:real}->{x:real,y:real}` can be passed for the first argument to `distMoved`.

First, it is safe to pass in a function with a return type that "promises" more, i.e., returns a subtype of the needed return type for the function `{x:real,y:real}`. For example, it is fine for this call to type-check:

```
fun flipGreen p = {x=~p.x, y=~p.y, color="green"}
val d = distMoved(flipGreen, {x=3.0, y=4.0})
```

The type of `flipGreen` is

```
{x:real,y:real} -> {x:real,y:real,color:string}
```

This is safe because `distMoved` expects a `{x:real,y:real}->{x:real,y:real}` and `flipGreen` is substitutable for values of such a type since the fact that `flipGreen` returns a record that also has a `color` field is not a problem.

In general, the rule here is that if `ta <: tb`, then `t -> ta <: t -> tb`, i.e., the subtype can have a return type that is a subtype of the supertype's return type. To introduce a little bit of jargon, we say return types are *covariant* for function subtyping meaning the subtyping for the return types works "the same way" (co) as for the types overall.

Now let us consider passing in a function with a different argument type. It turns out argument types are NOT covariant for function subtyping. Consider this example call to `distMoved`:

```
fun flipIfGreen p = if p.color = "green"
                    then {x=~p.x, y=~p.y}
                    else {x=p.x, y=p.y}
val d = distMoved(flipIfGreen, {x=3.0, y=4.0})
```

The type of `flipIfGreen` is

```
{x:real,y:real,color:string} -> {x:real,y:real}
```

This program should not type-check: If we run it, the expression `p.color` will have a "no such field" error since the point passed to `flipIfGreen` does not have a `color` field. In short, `ta <: tb`, does NOT mean `ta -> t <: tb -> t`. This would amount to using a function that "needs more of its argument" in place of a function that "needs less of its argument." This breaks the type system since the typing rules will not require the "more stuff" to be provided.

But it turns out it works just fine to use a function that "needs less of its argument" in place of a function that "needs more of its argument." Consider this example use of `distMoved`:

```
fun flipX_Y0 p = {x=~p.x, y=0.0}
val d = distMoved(flipX_Y0, {x=3.0, y=4.0})
```

The type of `flipX_Y0` is

```
{x:real} -> {x:real,y:real}
```

since the only field the argument to `flipX_Y0` needs is `x`. And the call to `distMoved` causes no problem: `distMoved` will always call its `f` argument with a record that has an `x` field and a `y` field, which is more than `flipX_Y0` needs.

In general, the treatment of argument types for function subtyping is "backwards" as follows: If `tb <: ta`, then `ta -> t <: tb -> t`. The technical jargon for "backwards" is *contravariance*, meaning the subtyping for argument types is the reverse (contra) of the subtyping for the types overall.

As a final example, function subtyping can allow contravariance of arguments and covariance of results:

```
fun flipXMakeGreen p = {x=~p.x, y=0.0, color="green"}
val d = distMoved(flipXMakeGreen, {x=3.0, y=4.0})
```

Here `flipXMakeGreen` has type

```
{x:real} -> {x:real,y:real,color:string}
```

This is a subtype of

```
{x:real,y:real} -> {x:real,y:real}
```

because `{x:real,y:real} <: {x:real}` (contravariance on arguments) and
`{x:real,y:real,color:string} <: {x:real,y:real}` (covariance on results).

The general rule for function subtyping is: If `t3 <: t1` and `t2 <: t4`, then `t1->t2 <: t3->t4`. This rule, combined with reflexivity (every type is a subtype of itself) lets us use contravariant arguments, covariant results, or both.

Argument contravariance is the least intuitive concept in the course, but it is worth burning into your memory so that you do not forget it. Many very smart people get confused because it is *not* about calls to methods/functions. Rather it is about the methods/functions themselves. We do not need function subtyping for passing non-function arguments to functions: we can just use other subtyping rules (e.g., those for records). Function subtyping is needed for higher-order functions or for storing functions themselves in records. And object types are related to having records with functions (methods) in them.

Subtyping for OOP

As promised, we can now apply our understanding of subtyping to OOP languages like Java or C#.

An object is basically a record holding fields (which we assume here are mutable) and methods. We assume the "slots" for methods are immutable: If an object's method `m` is implemented with some code, then there is no way to mutate `m` to refer to different code. (An instance of a subclass could have different code for `m`, but that is different than mutating a record field.)

With this perspective, sound subtyping for objects follows from sound subtyping for records and functions:

- A subtype can have extra fields.
- Because fields are mutable, a subtype cannot have a different type for a field.
- A subtype can have extra methods.
- Because methods are immutable, a subtype can have a subtype for a method, which means the method in the subtype can have contravariant argument types and a covariant result type.

That said, object types in Java and C# do not look like record types and function types. For example, we cannot write down a type that looks something like:

```
{fields : x:real, y:real, ...
 methods: distToOrigin : () -> real, ...}
```

Instead, we reuse class names as types where if there is a class `Foo`, then the type `Foo` includes in it all fields and methods implied by the class definition (including superclasses). And, as discussed previously, we also have interfaces, which are more like record types except they do not include fields and we use the name of the interface as a type. Subtyping in Java and C# includes only the subtyping explicitly stated via the subclass relationship and the interfaces that classes explicitly indicate they implement (including interfaces implemented by superclasses).

All said, this approach is more restrictive than subtyping requires, but since it does not allow anything it should not, it soundly prevents "field missing" and "method missing" errors. In particular:

- A subclass can add fields but not remove them
- A subclass can add methods but not remove them
- A subclass can override a method with a covariant return type
- A class can implement more methods than an interface requires or implement a required method with a covariant return type

Classes and types are different things! Java and C# purposely confuse them as a matter of convenience, but you should keep the concepts separate. A class defines an object's behavior. Subclassing inherits behavior, modifying behavior via extension and override. A type describes what fields an object has and what messages it can respond to. Subtyping is a question of substitutability and what we want to flag as a type error. So try to avoid saying things like, "overriding the method in the supertype" or, "using subtyping to pass an argument of the superclass." That said, this confusion is understandable in languages where every class declaration introduces a class and a type with the same name.

Covariant `self`/`this`

As a final subtle detail and advanced point, Java's `this` (i.e., Ruby's `self`) is treated specially from a type-checking perspective. When type-checking a class `C`, we know `this` will have type `C` or a subtype, so it is sound to assume it has type `C`. In a subtype, e.g., in a method overriding a method in `C`, we can assume `this` has the subtype. None of this causes any problems, and it is essential for OOP. For example, in class `B` below, the method `m` can type-check only if `this` has type `B`, not just `A`.

```
class A {
  int m(){ return 0; }
}
class B extends A {
  int x;
  int m(){ return x; }
}
```

But if you recall our manual encoding of objects in Racket, the encoding passed `this` as an extra explicit *argument* to a method. That would suggest *contravariant* subtyping, meaning `this` in a subclass could not have a *subtype*, which it needs to have in the example above.

It turns out `this` is special in the sense that while it is like an extra argument, it is an argument that is covariant. How can this be? Because it is not a "normal" argument where callers can choose "anything" of the correct type. Methods are always called with a `this` argument that is a subtype of the type the method expects.

This is the main reason why coding up dynamic dispatch manually works much less well in statically typed languages, even if they have subtyping: You need special support in your type system for `this`.

Generics Versus Subtyping

We have now studied both subtype polymorphism, also known as subtyping, and parametric polymorphism, also known as generic types, or just generics. So let's compare and contrast the two approaches, demonstrating what each is designed for.

What are generics good for?

There are many programming idioms that use generic types. We do not consider all of them here, but let's reconsider probably the two most common idioms that came up when studying higher-order functions.

First, there are functions that combine other functions such as `compose`:

```
val compose : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
```

Second, there are functions that operate over collections/containers where different collections/containers can hold values of different types:

```
val length : 'a list -> int
val map : ('a -> 'b) -> 'a list -> 'b list
val swap : ('a * 'b) -> ('b * 'a)
```

In all these cases, the key point is that if we had to pick non-generic types for these functions, we would end up with significantly less code reuse. For example, we would need one `swap` function for producing an `int * bool` from a `bool * int` and another `swap` function for swapping the positions of an `int * int`.

Generic types are much more useful and precise than just saying that some argument can "be anything." For example, the type of `swap` indicates that the second component of the result has the same type as the first component of the argument and the first component of the result has the same type as the second component of the argument. In general, we reuse a type variable to indicate when multiple things can have any type but must be the same type.

Generics in Java

Java has had subtype polymorphism since its creation in the 1990s and has had parametric polymorphism since 2004. Using generics in Java can be more cumbersome without ML's support for type inference and, as a separate matter, closures, but generics are still useful for the same programming idioms. Here, for example, is a generic `Pair` class, allowing the two fields to have any type:

```
class Pair<T1,T2> {
  T1 x;
  T2 y;
  Pair(T1 _x, T2 _y){ x = _x; y = _y; }
  Pair<T2,T1> swap() {
     return new Pair<T2,T1>(y,x);
  }
  ...
}
```

Notice that, analogous to ML, "`Pair`" is not a type: something like `Pair<String,Integer>` is a type. The `swap` method is, in object-oriented style, an instance method in `Pair<T1,T2>` that returns a `Pair<T2,T1>`. We could also define a static method:

```
  static <T1,T2> Pair<T2,T1> swap(Pair<T1,T2> p) {
     return new Pair<T2,T1>(p.y,p.x);
  }
```

For reasons of backwards-compatibility, the previous paragraph is not quite true: Java also has a type `Pair` that "forgets" what the types of its fields are. Casting to and from this "raw" type leads to compile-time warnings that you would be wise to heed: Ignoring them can lead to run-time errors in places you would not expect.

Subtyping is a Bad Substitute for Generics

If a language does not have generics or a programmer is not comfortable with them, one often sees generic code written in terms of subtyping instead. Doing so is like painting with a hammer instead of a paintbrush: technically possible, but clearly the wrong tool. Consider this Java example:

```
class LamePair {
  Object x;
  Object y;
  LamePair(Object _x, Object _y){ x=_x; y=_y; }
  LamePair swap() { return new LamePair(y,x); }
  ...
}

String s = (String)(new LamePair("hi",4).y); // error caught only at run-time
```

The code in LamePair type-checks without problem: the fields x and y have type Object, which is a supertype of every class and interface. The difficulties arise when clients use this class. Passing arguments to the constructor works as expected with subtyping.[3] But when we retrieve the contents of a field, getting an Object is not very useful: we want the type of value we put back in.

Subtyping does not work that way: the type system knows only that the field holds an Object. So we have to use a *downcast*, e.g., (String)e, which is a run-time check that the result of evaluating e is actually of type String, or, in general, a subtype thereof. Such run-time checks have the usual dynamic-checking costs in terms of performance, but, more importantly, in terms of the possibility of failure: this is not checked statically. Indeed, in the example above, the downcast would fail: it is the x field that holds a String, not the y field.

In general, when you use Object and downcasts, you are essentially taking a dynamic typing approach: any object could be stored in an Object field, so it is up to programmers, without help from the type system, to keep straight what kind of data is where.

What is Subtyping Good For?

We do not suggest that subtyping is not useful: It is great for allowing code to be reused with data that has "extra information." For example, geometry code that operates over points should work fine for colored-points. It is certainly inconvenient in such situations that ML code like this simply does not type-check:

```
fun distToOrigin1 {x=x,y=y} =
   Math.sqrt (x*x + y*y)

(* does not type-check *)
(* val five = distToOrigin1 {x=3.0,y=4.0,color="red"} *)
```

A generally agreed upon example where subtyping works well is graphical user interfaces. Much of the code for graphics libraries works fine for any sort of graphical element ("paint it on the screen," "change the background color," "report if the mouse is clicked on it," etc.) where different elements such as buttons, slider bars, or text boxes can then be subtypes.

Generics are a Bad Substitute for Subtyping

In a language with generics instead of subtyping, you can code up your own code reuse with higher-order functions, but it can be quite a bit of trouble for a simple idea. For example, distToOrigin2 below uses getters passed in by the caller to access the x and y fields and then the next two functions have different types but identical bodies, just to appease the type-checker.

```
fun distToOrigin2(getx,gety,v) =
    let
        val x = getx v
```

[3] Java will automatically convert a 4 to an Integer object holding a 4.

```
        val y = gety v
    in
        Math.sqrt (x*x + y*y)
    end

fun distToOriginPt (p : {x:real,y:real}) =
    distToOrigin2(fn v => #x v,
                  fn v => #y v,
                  p)

fun distToOriginColorPt (p : {x:real,y:real,color:string}) =
    distToOrigin2(fn v => #x v,
                  fn v => #y v,
                  p)
```

Nonetheless, without subtyping, it may sometimes be worth writing code like `distToOrigin2` if you want it to be more reusable.

Bounded Polymorphism

As Java and C# demonstrate, there is no reason why a statically typed programming language cannot have generic types and subtyping. There are some complications from having both that we will not discuss (e.g., static overloading and subtyping are more difficult to define), but there are also benefits. In addition to the obvious benefit of supporting separately the idioms that each feature supports well, we can combine the ideas to get even more code reuse and expressiveness.

The key idea is to have *bounded generic types*, where instead of just saying “a subtype of `T`” or “for all types `'a`,” we can say, “for all types `'a` that are a subtype of `T`.” Like with generics, we can then use `'a` multiple times to indicate where two things must have the same type. Like with subtyping, we can treat `'a` as a subtype of `T`, accessing whatever fields and methods we know a `T` has.

We will show an example using Java, which hopefully you can follow just by knowing that `List<Foo>` is the syntax for the type of lists holding elements of type `Foo`.

Consider this `Point` class with a `distance` method:

```
class Pt {
    double x, y;
    double distance(Pt pt) { return Math.sqrt((x-pt.x)*(x-pt.x)+(y-pt.y)*(y-pt.y));  }
    Pt(double _x, double _y) { x = _x; y = _y; }
}
```

Now consider this static method that takes a list of points `pts`, a point `center`, and a radius `radius` and returns a new list of points containing all the input points within `radius` of `center`, i.e., within the circle defined by `center` and `radius`:

```
static List<Pt> inCircle(List<Pt> pts, Pt center, double radius) {
    List<Pt> result = new ArrayList<Pt>();
    for(Pt pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      return result;
}
```

(Understanding the code in the method body is not important.)

This code works perfectly fine for a `List<Pt>`, but if `ColorPt` is a subtype of `Pt` (adding a `color` field and associated methods), then we cannot call `inCircle` method above with a `List<ColorPt>` argument. Because depth subtyping is unsound with mutable fields, `List<ColorPt>` is not a subtype of `List<Pt>`. Even if it were, we would like to have a result type of `List<ColorPt>` when the argument type is `List<ColorPt>`.

For the code above, this is true: If the argument is a `List<ColorPt>`, then the result will be too, but we want a way to express that in the type system. Java's bounded polymorphism lets us describe this situation (the syntax details are not important):

```
static <T extends Pt> List<T> inCircle(List<T> pts, Pt center, double radius) {
  List<T> result = new ArrayList<T>();
    for(T pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      return result;
}
```

This method is polymorphic in type `T`, but `T` must be a subtype of `Pt`. This subtyping is necessary so that the method body can call the `distance` method on objects of type `T`. Wonderful!

Optional: Additional Java-Specific Bounded Polymorphism

While the second version of `inCircle` above is ideal, let us now consider a few variations. First, Java does have enough dynamically checked casts that it is possible to use the first version with a `List<ColorPt>` argument and cast the result from `List<Pt>` to `List<ColorPt>`. We have to use the "raw type" `List` to do it, something like this where `cps` has type `List<ColorPt>`.

```
List<ColorPt> out = (List<ColorPt>)(List) inCircle((List<Pt>)(List)cps, new Pt(0.0,0.0), 1.5);
```

In this case, these casts turn out to be okay: if `inCircle` is passed a `List<ColorPt>` the result will be a `List<ColorPt>`. But casts like this are dangerous. Consider this variant of the method that has the same type as the initial non-generic `inCircle` method:

```
static List<Pt> inCircle(List<Pt> pts, Pt center, double radius) {
    List<Pt> result = new ArrayList<Pt>();
    for(Pt pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      else
        result.add(center);
      return result;
}
```

The difference is that any points not within the circle are "replaced" in the output by `center`. Now if we call `inCircle` with a `List<ColorPt> cps` where one of the points is not within the circle, then the result is *not* a List<ColorPt> — it contains a Pt object! You might expect then that the cast of the result to List<ColorPt> would fail, but Java does not work this way for backward-compatibility reasons: even this cast succeeds. So now we have a value of type List<ColorPt> that is not a list of ColorPt objects. What happens instead in Java is that a cast will fail later when we get a value from this alleged List<ColorPt> and try to use it as ColorPt when it is in fact a Pt. The blame is clearly in the wrong place, which is why using the warning-inducing casts in the first place is so problematic.

Last, we can discuss what type is best for the center argument in our bounded-polymorphic version. Above, we chose Pt, but we could also choose T:

```
static <T extends Pt> List<T> inCircle(List<T> pts, T center, double radius) {
    List<T> result = new ArrayList<T>();
    for(T pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      return result;
}
```

It turns out this version allows *fewer* callers since the previous version allows, for example, a first argument of type List<ColorPt> and a second argument of type Pt (and, therefore, via subtyping, also a ColorPt). With the argument of type T, we require a ColorPt (or a subtype) when the first argument has type List<ColorPt>. On the other hand, our version that sometimes adds center to the output requires the argument to have type T:

```
static <T extends Pt> List<T> inCircle(List<T> pts, T center, double radius) {
    List<T> result = new ArrayList<T>();
    for(T pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      else
        result.add(center);
      return result;
}
```

In this last version, if center has type Pt, then the call result.add(center) does not type-check since Pt may not be a subtype of T (what we know is T is a subtype of Pt). The actual error message may be a bit confusing: It reports there is no add method for List<T> that takes a Pt, which is true: the add method we are trying to use takes a T.