Programming Language
Part C

Coursera - University of Washington

October 5, 2016

Coursera Programming Languages Course
Section 8 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

Ruby Logistics 1
Ruby Features Most Interesting for a PL Course 2
The Rules of Class-Based OOP 3
Objects, Classes, Methods, Variables, Etc. 3
Visibility and Getters/Setters 6
Some Syntax, Semantics, and Scoping To Get Used To 7
Everything is an Object 7
The Top-Level 8
Class Definitions are Dynamic 8
Duck Typing 8
Arrays 9
Passing Blocks 11
Using Blocks 12
The `Proc` Class 12
Hashes and Ranges 13
Subclassing and Inheritance 14
Why Use Subclassing? 16
Overriding and Dynamic Dispatch 17
The Precise Definition of Method Lookup 19
Dynamic Dispatch Versus Closures 20
Optional: Implementing Dynamic Dispatch Manually in Racket 21

Ruby Logistics

The course website provides installation and basic usage instructions for Ruby and its REPL (called `irb`), so that information is not repeated here. Note that for consistency we will require Ruby version 2.x.y (for any x and y), although this is for homework purposes – the concepts we will discuss do not depend on an exact version, naturally.

There is a great amount of free documentation for Ruby at `http://ruby-doc.org` and `http://www.ruby-lang.org/en/documentation/`. We also recommend, *Programming Ruby 1.9 & 2.0, The Pragmatic Programmers' Guide* although this book is not free. Because the online documentation is excellent, the other course materials may not describe in detail every language feature used in the lectures and homeworks although it is also not our goal to make you hunt for things on purpose. In general, learning new language features and libraries is an important skill after some initial background to point you in the right direction.

Ruby Features Most Interesting for a PL Course

Ruby is a large, modern programming language with various features that make it popular. Some of these features are useful for a course on programming-language features and semantics, whereas others are not useful for our purposes even though they may be very useful in day-to-day programming. Our focus will be on object-oriented programming, dynamic typing, blocks (which are almost closures), and mixins. We briefly describe these features and some other things that distinguish Ruby here — if you have not seen an object-oriented programming language, then some of this overview will not make sense until after learning more Ruby.

- Ruby is a *pure object-oriented* language, which means *all* values in the language are objects. In Java, as an example, some values that are not objects are `null`, `13`, `true`, and `4.0`. In Ruby, every expression evaluates to an object.
- Ruby is *class-based*: Every object is an instance of a class. An object's class determines what methods an object has. (All code is in methods, which are like functions in the sense that they take arguments and return results.) You call a method "on" an object, e.g., `obj.m(3,4)` evaluates the variable `obj` to an object and calls its `m` method with arguments 3 and 4. Not all object-oriented languages are class-based; see, for example, JavaScript.
- Ruby has *mixins*: The next module will describe mixins, which strike a reasonable compromise between multiple inheritance (like in C++) and interfaces (like in Java). Every Ruby class has one superclass, but it can include any number of mixins, which, unlike interfaces, can define methods (not just require their existence).
- Ruby is *dynamically typed*: Just as Racket allowed calling any function with any argument, Ruby allows calling any method on any object with any arguments. If the *receiver* (the object on which we call the method) does not define the method, we get a dynamic error.
- Ruby has *many dynamic features*: In addition to dynamic typing, Ruby allows instance variables (called fields in many object-oriented languages) to be added and removed from objects and it allows methods to be added and removed from classes while a program executes.
- Ruby has *convenient reflection*: Various built-in methods make it easy to discover at run-time properties about objects. As examples, every object has a method `class` that returns the object's class, and a method `methods` that returns an array of the object's methods.
- Ruby has *blocks* and *closures*: Blocks are almost like closures and are used throughout Ruby libraries for convenient higher-order programming. Indeed, it is rare in Ruby to use an explicit loop since collection classes like `Array` define so many useful iterators. Ruby also has fully-powerful closures for when you need them.
- Ruby is a *scripting language*: There is no precise definition of a what makes a language a scripting language. It means the language is engineered toward making it easy to write short programs, providing convenient access to manipulating files and strings (topics we will not discuss), and having less concern for performance. Like many scripting languages, Ruby does not require that you declare variables before using them and there are often many ways to say the same thing.
- Ruby is *popular for web applications*: The Ruby on Rails framework is a popular choice for developing the server side of modern web-sites.

Recall that, taken together, ML, Racket, and Ruby cover three of the four combinations of functional vs. object-oriented and statically vs. dynamically typed.

Our focus will be on Ruby's object-oriented nature, not on its benefits as a scripting language. We also will not discuss at all its support for building web applications, which is a main reason it is currently so popular. As an object-oriented language, Ruby shares much with Smalltalk, a language that has basically not changed since 1980. Ruby does have some nice additions, such as mixins.

Ruby is also a large language with a "why not" attitude, especially with regard to syntax. ML and Racket (and Smalltalk) adhere rather strictly to certain traditional programming-language principles, such as defining a small language with powerful features that programmers can then use to build large libraries. Ruby often takes the opposite view. For example, there are many different ways to write an if-expression.

The Rules of Class-Based OOP

Before learning the syntax and semantics of particular Ruby constructs, it is helpful to enumerate the "rules" that describe languages like Ruby and Smalltalk. Everything in Ruby is described in terms of *object-oriented programming*, which we abbreviate OOP, as follows:

1. All values (as usual, the result of evaluating expressions) are references to *objects*.
2. Given an object, code "communicates with it" by calling its *methods*. A synonym for calling a method is *sending a message*. (In processing such a message, an object is likely to send other messages to other objects, leading to arbitrarily sophisticated computations.)
3. Each object has its own private *state*. Only an object's methods can directly access or update this state.
4. Every object is an instance of a *class*.
5. An object's class determines the object's *behavior*. The class contains method definitions that dictate how an object handles method calls it receives.

While these rules are mostly true in other OOP languages like Java or C#, Ruby makes a more complete commitment to them. For example, in Java and C#, some values like numbers are not objects (violating rule 1) and there are ways to make object state publicly visible (violating rule 3).

Objects, Classes, Methods, Variables, Etc.

(See also the example programs posted with the lecture materials, not all of which are repeated here.)

Class and method definitions

Since every *object* has a *class*, we need to define classes and then create *instances* of them (an object of class `C` is an instance of `C`). (Ruby also predefines many classes in its language and standard library.) The basic syntax (we will add features as we go) for creating a class `Foo` with *methods* `m1`, `m2`, ... `mn` can be:

```
class Foo
   def m1
     ...
   end

   def m2 (x,y)
     ...
```

```
  end

  ...

  def mn z
    ...
  end
end
```

Class names must be capitalized. They include method definitions. A method can take any number of arguments, including 0, and we have a variable for each argument. In the example above, `m1` takes 0 arguments, `m2` takes two arguments, and `mn` takes 1 argument. Not shown here are method bodies. Like ML and Racket functions, a method implicitly returns its last expression. Like Java/C#/C++, you can use an explicit `return` statement to return immediately when helpful. (It is bad style to have a return at the end of your method since it can be implicit there.)

Method arguments can have defaults in which case a caller can pass fewer actual arguments and the remaining ones are filled in with defaults. If a method argument has a default, then all arguments to its right must also have a default. An example is:

```
  def myMethod (x,y,z=0,w="hi")
     ...
  end
```

Calling methods

The method call `e0.m(e1, ..., en)` evaluates `e0`, `e1`, ..., `en` to objects. It then calls the method `m` in the result of `e0` (as determined by the class of the result of `e0`), passing the results of `e1`, ..., `en` as arguments. As for syntax, the parentheses are optional. In particular, a zero-argument call is usually written `e0.m`, though `e0.m()` also works.

To call another method on the same object as the currently executing method, you can write `self.m(...)` or just `m(...)`. (Java/C#/C++ work the same way except they use the keyword `this` instead of `self`.)

In OOP, another common name for a method call is a *message send*. So we can say `e0.m e1` sends the result of `e0` the message `m` with the argument that is the result of `e1`. This terminology is “more object-oriented” — as a client, we do not care how the receiver (of the message) is implemented (e.g., with a method named `m`) as long as it can handle the message. As general terminology, in the call `e0.m args`, we call the result of evaluating `e0` the *receiver* (the object receiving the message).

Instance variables

An object has a class, which defines its methods. It also has *instance variables*, which hold values (i.e., objects). Many languages (e.g., Java) use the term *fields* instead of instance variables for the same concept. Unlike Java/C#/C++, our class definition does not indicate what instance variables an instance of the class will have. To add an instance variable to an object, you just assign to it: if the instance variable does not already exist, it is created. All instance variables start with an `@`, e.g., `@foo`, to distinguish them from variables local to a method.

Each object has its own instance variables. Instance variables are mutable. An expression (in a method body) can read an instance variable with an expression like `@foo` and write an instance variable with an expression `@foo = newValue`. Instance variables are private to an object. There is no way to directly access an instance variable of any other object. So `@foo` refers to the `@foo` instance variable of the current object, i.e., `self.@foo` except `self.@foo` is *not* actually legal syntax.

Ruby also has class variables (which are like Java's static fields). They are written `@@foo`. Class variables are not private to an object. Rather, they are *shared* by all instances of the class, but are still not directly accessible from objects of different classes.

Constructing an object

To create a new instance of class `Foo`, you write `Foo.new (...)` where `(...)` holds some number of arguments (where, as with all method calls, the parentheses are optional and when there are zero or one arguments it is preferred to omit them). The call to `Foo.new` will create a new instance of `Foo` and then, before `Foo.new` returns, call the new object's `initialize` method with all the arguments passed to `Foo.new`. That is, the method `initialize` is special and serves the same role as constructors in other object-oriented languages.

Typical behavior for `initialize` is to create and initialize instance variables. In fact, the normal approach is for `initialize` always to create the same instance variables and for no other methods in the class to create instance variables. But Ruby does not require this and it may be useful on occasion to violate these conventions. Therefore, different instances of a class can have different instance variables.

Expressions and Local Variables

Most expressions in Ruby are actually method calls. Even `e1 + e2` is just syntactic sugar for `e1.+ e2`, i.e., call the `+` method on the result of `e1` with the result of `e2`. Another example is `puts e`, which prints the result of `e` (after calling its `to_s` method to convert it to a string) and then a newline. It turns out `puts` is a method in all objects (it is defined in class `Object` and all classes are subclasses of `Object` — we discuss subclasses later), so `puts e` is just `self.puts e`.

Not every expression is a method call. The most common other expression is some form of conditional. There are various ways to write conditionals; see the example code posted with the lecture materials. As discussed below, loop expressions are rare in Ruby code.

Like instance variables, variables local to a method do not have to be declared: The first time you assign to `x` in a method will create the variable. The scope of the variable is the entire method body. It is a run-time error to use a local variable that has not yet been defined. (In contrast, it is not a run-time error to use an instance variable that has not yet been defined. Instead you get back the `nil` object, which is discussed more below.)

Class Constants and Class Methods

A class constant is a lot like a class variable (see above) except that (1) it starts with a capital letter instead of `@@`, (2) you should not mutate it, and (3) it is publicly visible. Outside of an instance of class `C`, you can access a constant `Foo` of `C` with the syntax `C::Foo`. An example is `Math::PI`.[1]

A class method is like an ordinary method (called an instance method to distinguish from class methods) except (1) it does not have access to any of the instance variables or instance methods of an instance of the class and (2) you can call it from outside the class `C` where it is defined with `C.method_name args`. There are various ways to define a class method; the most common is the somewhat hard-to-justify syntax:

```
def self.method_name args
  ...
end
```

Class methods are called static methods in Java and C#.

[1]Actually, `Math` is a module, not a class, so this is not technically an example, but modules can also have constants.

Visibility and Getters/Setters

As mentioned above, instance variables are private to an object: only method calls with *that object* as the receiver can read or write the fields. As a result, the syntax is `@foo` and the self-object is implied. Notice even other instances of the same class cannot access the instance variables. This is quite object-oriented: you can interact with another object only by sending it messages.

Methods can have different *visibilities*. The default is `public`, which means any object can call the method. There is also `private`, which, like with instance variables, allows only the object itself to call the method (from other methods in the object). In-between is `protected`: A protected method can be called by any object that is an instance of the same class or any subclass of the class.

There are various ways to specify the visibility of a method. Perhaps the simplest is within the class definition you can put `public`, `private`, or `protected` between method definitions. Reading top-down, the most recent visibility specified holds for all methods until the next visibility is specified. There is an implicit `public` before the first method in the class.

To make the contents of an instance variable available and/or mutable, we can easily define getter and setter methods, which by convention we can give the same name as the instance variable. For example:

```
def foo
  @foo
end

def foo= x
  @foo = x
end
```

If these methods are public, now any code can access the instance variable `@foo` indirectly, by calling `foo` or `foo=`. It sometimes makes sense to instead make these methods `protected` if only other objects of the same class (or subclasses) should have access to the instance variables.

As a cute piece of syntactic sugar, when calling a method that ends in a `=` character, you can have spaces before the `=`. Hence you can write `e.foo = bar` instead of `e.foo= bar`.

The advantage of the getter/setter approach is it remains an implementation detail that these methods are implemented as getting and setting an instance variable. We, or a subclass implementer, could change this decision later without clients knowing. We can also omit the setter to ensure an instance variable is not mutated except perhaps by a method of the object.

As an example of a "setter method" that is not *actually* a setter method, a class could define:

```
def celsius_temp= x
  @kelvin_temp = x + 273.15
end
```

A client would likely imagine the class has a `@celsius_temp` instance variable, but in fact it (presumably) does not. This is a good abstraction that allows the implementation to change.

Because getter and setter methods are so common, there is shorter syntax for defining them. For example, to define getters for instance variables `@x`, `@y`, and `@z` and a setter for `@x`, the class definition can just include:

```
attr_reader :y, :z # defines getters
attr_accessor :x # defines getters and setters
```

A final syntactic detail: If a method `m` is private, you can only call it as `m` or `m(args)`. A call like `x.m` or `x.m(args)` would break visibility rules. A call like `self.m` or `self.m(args)` would not break visibility, but still is not allowed.

Some Syntax, Semantics, and Scoping To Get Used To

Ruby has a fair number of quirks that are often convenient for quickly writing useful programs but may take some getting used to. Here are some examples; you will surely discover more.

- There are several forms of conditional expressions, including `e1 if e2` (all on one line), which evaluates `e1` only if `e2` is true (i.e., it reads right-to-left).
- Newlines are often significant. For example, you can write

  ```
  if e1
    e2
  else
    e3
  end
  ```

  But if you want to put this all on one line, then you need to write `if e1 then e2 else e3 end`. Note, however, indentation is never significant (only a matter of style).
- Conditionals can operate on any object and treat every object as "true" with *two* exceptions: `false` and `nil`.
- As discussed above, you can define a method with a name that ends in `=`, for example:

  ```
  def foo= x
    @blah = x * 2
  end
  ```

  As expected, you can write `e.foo=(17)` to change `e`'s `@blah` instance variable to be 34. Better yet, you can adjust the parentheses and spacing to write `e.foo = 17`. This is just syntactic sugar. It "feels" like an assignment statement, but it is really a method call. Stylistically you do this for methods that mutate an object's state in some "simple" way (like setting a field).
- Where you write `this` in Java/C#/C++, you write `self` in Ruby.
- Remember variables (local, instance, or class) get automatically created by assignment, so if you misspell a variable in an assignment, you end up just creating a different variable.

Everything is an Object

Everything is an object, including numbers, booleans, and `nil` (which is often used like `null` in Java). For example, `-42.abs` evaluates to `42` because the `Fixnum` class defines the method `abs` to compute the absolute value and `-42` is an instance of `Fixnum`. (Of course, this is a silly expression, but `x.abs` where `x` currently holds `-42` is reasonable.)

All objects have a `nil?` method, which the class of `nil` defines to return `true` but other classes define to return `false`. Like in ML and Racket, every expression produces a result, but when no particular result makes sense, `nil` is preferred style (much like ML's `()` and Racket's void-object). That said, it is often convenient for methods to return `self` so that subsequent method calls to the same object can be put together. For example, if the `foo` method returns `self`, then you can write `x.foo(14).bar("hi")` instead of

```
x.foo(14)
x.bar("hi")
```

There are many methods to support *reflection* — learning about objects and their definition during program execution — that are defined for all objects. For example, the method `methods` returns an array of the names of the methods defined on an object and the method `class` returns the class of the object.[2] Such reflection is occasionally useful in writing flexible code. It is also useful in the REPL or for debugging.

The Top-Level

You can define methods, variables, etc. outside of an explicit class definition. The methods are implicitly added to class `Object`, which makes them available from within any object's methods. Hence all methods are really part of some class.[3]

Top-level expressions are evaluated in order when the program runs. So instead of Ruby specifying a main class and method with a special name (like `main`), you can just create an object and call a method on it at top-level.

Class Definitions are Dynamic

A Ruby program (or a user of the REPL) can change class definitions while a Ruby program is running. Naturally this affects all users of the class. Perhaps surprisingly, it even affects instances of the class that have already been created. That is, if you create an instance of `Foo` and then add or delete methods in `Foo`, then the already-created object "sees" the changes to its behavior. After all, every object has a class and the (current) class (definition) defines an object's behavior.

This is usually dubious style because it breaks abstractions, but it leads to a simpler language definition: defining classes and changing their definitions is just a run-time operation like everything else. It can certainly break programs: If I change or delete the `+` method on numbers, I would not expect many programs to keep working correctly. It can be useful to add methods to existing classes, especially if the designer of the class did not think of a useful helper method.

The syntax to add or change methods is particularly simple: Just give a class definition including method definitions for a class that is already defined. The method definitions either replace definitions for methods previously defined (with the same name method name) or are added to the class (if no method with the name previously existed).

Duck Typing

Duck typing refers to the expression, "If it walks like a duck and quacks like a duck, then it's a duck" though a better conclusion might be, "then there is no reason to concern yourself with the possibility that it might

[2]This class is itself just another object. Yes, even classes are objects.
[3]This is not entirely true because modules are not classes.

not be a duck." In Ruby, this refers to the idea that the class of an object (e.g., "Duck") passed to a method is not important so long as the object can respond to all the messages it is expected to (e.g., "walk to x" or "quack now").

For example, consider this method:

```
def mirror_update pt
  pt.x = pt.x * -1
end
```

It is natural to view this as a method that must take an instance of a particular class `Point` (not shown here) since it uses methods `x` and `x=` defined in it. And the `x` getter must return a number since the result of `pt.x` is sent the `*` message with `-1` for multiplication.

But this method is more generally useful. It is not necessary for `pt` to be an instance of `Point` provided it has methods `x` and `x=`.

Moreover, the `x` and `x=` methods need not be a getter and setter for an instance variable `@x`.

Even more generally, we do not need the `x` method to return a number. It just has to return some object that can respond to the `*` message with argument `-1`.

Duck typing can make code more reusable, allowing clients to make "fake ducks" and still use your code. In Ruby, duck typing basically "comes for free" as long you do not explicitly check that arguments are instances of particular classes using methods like `instance_of?` or `is_a?` (discussed below when we introduce subclassing).

Duck typing has disadvantages. The most lenient specification of how to use a method ends up describing the whole implementation of a method, in particular what messages it sends to what objects. If our specification reveals all that, then almost no variant of the implementation will be equivalent. For example, if we know `i` is a number (and ignoring clients redefining methods in the classes for numbers), then we can replace `i+i` with `i*2` or `2*i`. But if we just assume `i` can receive the `+` message with itself as an argument, then we cannot do these replacements since `i` may not have a `*` method (breaking `i*2`) or it may not be the sort of object that `2` expects as an argument to `*` (breaking `2*i`).

Arrays

The `Array` class is *very* commonly used in Ruby programs and there is special syntax that is often used with it. Instances of `Array` have all the uses that arrays in other programming languages have — and much, much more. Compared to arrays in Java/C#/C/etc., they are much more flexible and dynamic with fewer operations being errors. The trade-off is they can be less efficient, but this is usually not a concern for convenient programming in Ruby. In short, all Ruby programmers are familiar with Ruby arrays because they are the standard choice for any sort of collection of objects.

In general, an array is a mapping from numbers (the indices) to objects. The syntax `[e1,e2,e3,e4]` creates a new array with four objects in it: The result of `e1` is in index 0, the result of `e2` is in index 1, and so on. (Notice the indexing starts at 0.) There are other ways to create arrays. For example, `Array.new(x)` creates an array of length `x` with each index initially mapped to `nil`. We can also pass blocks (see below for what blocks actually are) to the `Array.new` method to initialize array elements. For example, `Array.new(x) { 0 }` creates an array of length `x` with all elements initialized to `0` and `Array.new(5) {|i| -i }` creates the array `[0,-1,-2,-3,-4]`.

The syntax for getting and setting array elements is similar to many other programming languages: The expression `a[i]` gets the element in index `i` of the array referred to by `a` and `a[i] = e` sets the same array index. As you might suspect in Ruby, we are really just calling methods on the `Array` class when we use this syntax.

Here are some simple ways Ruby arrays are more dynamic and less error-causing than you might expect compared to other programming languages:

- As usual in a dynamically typed language, an array can hold objects that are instances of different classes, for example `[14, "hi", false, 34]`.
- Negative array indices are interpreted from the *end* of the array. So `a[-1]` retrieves the last element in the array `a`, `a[-2]` retrieves the second-to-last element, etc.
- There are no array-bounds errors. For the expression `a[i]`, if `a` holds fewer than `i+1` objects, then the result will just be `nil`. Setting such an index is even more interesting: For `a[i]=e`, if `a` holds fewer than `i+1` objects, then the array will *grow dynamically* to hold `i+1` objects, the last of which will be the result of `e`, with the right number of `nil` objects between the old last element and the new last element.
- There are *many* methods and operations defined in the standard library for arrays. If the operation you need to perform on an array is at all general-purpose, peruse the documentation since it is surely already provided. As two examples, the `+` operator is defined on arrays to mean concatenation (a new array where all of the left-operand elements precede all of the right-operand elements), and the `|` operator is like the `+` operator except it removes all duplicate elements from the result.

In addition to all the conventional uses for arrays, Ruby arrays are also often used where in other languages we would use other constructs for tuples, stacks, or queues. Tuples are the most straightforward usage. After all, given dynamic typing and less concern for efficiency, there is little reason to have separate constructs for tuples and arrays. For example, for a triple, just use a 3-element array.

For stacks, the `Array` class defines convenient methods `push` and `pop`. The former takes an argument, grows the array by one index, and places the argument at the new last index. The latter shrinks the array by one index and returns the element that was at the old last index. Together, this is exactly the last-in-first-out behavior that defines the behavior of a stack. (How this is implemented in terms of actually growing and shrinking the underlying storage for the elements is of concern only in the implementation of `Array`.)

For queues, we can use `push` to add elements as just described and use the `shift` method to dequeue elements. The `shift` method returns the object at index 0 of the array, removes it from the array, and shifts all the other elements down one index, i.e., the object (if any) previously at index 1 is now at index 0, etc. Though not needed for simple queues, `Array` also has an `unshift` method that is like `push` except it puts the new object at index 0 and moves all other objects up by 1 index (growing the array size by 1).

Arrays are even more flexible than described here. For example, there are operations to replace any sequence of array elements with the elements of any other array, even if the other array has a different length than the sequence being replaced (hence changing the length of the array).

Overall, this flexible treatment of array sizes (growing and shrinking) is different from arrays in some other programming languages, but it is consistent with treating arrays as maps from numeric indices to objects.

What we have not shown so far are operations that perform some computation using all the contents of an array, such as mapping over the elements to make a new array, or computing a sum of them. That is because the Ruby idioms for such computations use *blocks*, which we introduce next.

Passing Blocks

While Ruby has while loops and for loops not unlike Java, most Ruby code does not use them. Instead, many classes have methods that take *blocks*. These blocks are *almost* closures. For example, integers have a `times` method that takes a block and executes it the number of times you would imagine. For example,

```
x.times { puts "hi" }
```

prints `"hi"` 3 times if `x` is bound to 3 in the environment.

Blocks are closures in the sense that they can refer to variables in scope where the block is defined. For example, after this program executes, `y` is bound to 10:

```
y = 7
[4,6,8].each { y += 1 }
```

Here `[4,6,8]` is an array with with 3 elements. Arrays have a method `each` that takes a block and executes it once for each element. Typically, however, we want the block to be passed each array element. We do that like this, for example to sum an array's elements and print out the running sum at each point:

```
sum = 0
[4,6,8].each { |x|
  sum += x
  puts sum
}
```

Blocks, surprisingly, are not objects. You cannot pass them as "regular" arguments to a method. Rather, any method can be passed either 0 or 1 blocks, separate from the other arguments. As seen in the examples above, the block is just put to the right of the method call. It is also after any other "regular" arguments. For example, the `inject` method is like the `fold` function we studied in ML and we can pass it an initial accumulator as a regular argument:

```
sum = [4,6,8].inject(0) { |acc,elt| acc + elt }
```

(It turns out the initial accumulator is optional. If omitted, the method will use the array element in index 0 as the initial accumulator.)

In addition to the braces syntax shown here, you can write a block using `do` instead of `{` and `end` instead of `}`. This is generally considered better style for blocks more than one line long.

When calling a method that takes a block, you should know how many arguments will be passed to the block when it is called. For the `each` method in `Array`, the answer is 1, but as the first example showed, you can ignore arguments if you have no need for them by omitting the `|...|`.

Many collections, including arrays, have a variety of block-taking methods that look very familiar to functional programmers, including `map`. As another example, the `select` method is like the function we called `filter`. Other useful *iterators* include `any?` (returns true if the block returns true for any element of the collection), `all?` (returns true if the block returns true for every element of the collection), and several more.

Using Blocks

While many uses of blocks involve calling methods in the standard library, you can also define your own methods that take blocks. (The large standard library just makes it somewhat rare to need to do this.)

You can pass a block to *any* method. The method body calls the block using the `yield` keyword. For example, this code prints `"hi"` 3 times:

```
def foo x
  if x
    yield
  else
    yield
    yield
  end
end
foo true { puts "hi" }
foo false { puts "hi" }
```

To pass arguments to a block, you put the arguments after the `yield`, e.g., `yield 7` or `yield(8,"str")`.

Using this approach, the fact that a method may expect a block is implicit; it is just that its body might use `yield`. An error will result if `yield` is used and no block was passed. The behavior when the block and the `yield` disagree on the number of arguments is somewhat flexible and not described in full detail here. A method can use the `block_given?` primitive to see if the caller provided a block. You are unlikely to use this method often: If a block is needed, it is conventional just to assume it is given and have `yield` fail if it is not. In situations where a method may or may not expect a block, often other regular arguments determine whether a block should be present. If not, then `block_given?` is appropriate.

Here is a recursive method that counts how many times it calls the block (with increasing numbers) before the block returns a true result.

```
def count i
    if yield i
      1
    else
      1 + (count(i+1) {|x| yield x})
    end
  end
```

The odd thing is that there is no direct way to pass the caller's block as the callee's block argument. But we can create a new block `{|x| yield x}` and the lexical scope of the `yield` in its body will do the right thing. If blocks were actually function closures that we could pass as objects, then this would be unnecessary function wrapping.

The `Proc` Class

Blocks are not quite closures because they are not objects. We cannot store them in a field, pass them as a regular method argument, assign them to a variable, put them in an array, etc. (Notice in ML and Racket, we could do the equivalent things with closures.) Hence we say that blocks are not "first-class values" because a first-class value is something that can be passed and stored like anything else in the language.

However, Ruby has “real” closures too: The class `Proc` has instances that are closures. The method `call` in `Proc` is how you apply the closure to arguments, for example `x.call` (for no arguments) or `x.call(3,4)`.

To make a `Proc` out of a block, you can write `lambda { ... }` where `{ ... }` is any block. Interestingly, `lambda` is not a keyword. It is just a method in class `Object` (and every class is a subclass of `Object`, so `lambda` is available everywhere) that creates a `Proc` out of a block it is passed. You can define your own methods that do this too; consult the documentation for the syntax to do this.

Usually all we need are blocks, such as in these examples that pass blocks to compute something about an array:

```
a = [3,5,7,9]
b = a.map {|x| x + 1}
i = b.count {|x| x >= 6}
```

But suppose we wanted to create an array of blocks, i.e., an array where each element was something we could “call” with a value. You cannot do this in Ruby because arrays hold objects and blocks are not objects. So this is an error:

```
c = a.map {|x| {|y| x >= y} } # wrong, a syntax error
```

But we can use `lambda` to create an array of instances of `Proc`:

```
c = a.map {|x| lambda {|y| x >= y} }
```

Now we can send the `call` message to elements of the `c` array:

```
c[2].call 17
j = c.count {|x| x.call(5) }
```

Ruby’s design is an interesting contrast from ML and Racket, which just provide full closures as the natural choice. In Ruby, blocks are more convenient to use than `Proc` objects and suffice in most uses, but programmers still have `Proc` objects when needed. Is it better to distinguish blocks from closures and make the more common case easier with a less powerful construct, or is it better just to have one general fully powerful feature?

Hashes and Ranges

The `Hash` and `Range` classes are two standard-library classes that are also very common but probably a little less common than arrays. Like arrays, there is special built-in syntax for them. They are also similar to arrays and support many of the same iterator methods, which helps us re-enforce the concept that “how to iterate” can be separated from “what to do while iterating.”

A hash is like an array except the mapping is not from numeric indices to objects. Instead, the mapping is from *(any) objects* to objects. If *a* maps to *b*, we call *a* a *key* and *b* a *value*. Hence a hash is a collection that maps a set of keys (all keys in a hash are distinct) to values, where the keys and values are just objects. We can create a hash with syntax like this:

```
{"SML" => 7, "Racket" => 12, "Ruby" => 42}
```

As you might expect, this creates a hash with keys that here are strings. It is also common (and more efficient) to use Ruby's symbols for hash keys as in:

```
{:sml => 7, :racket => 12, :ruby => 42}
```

We can get and set values in a hash using the same syntax as for arrays, where again the key can be anything, such as:

```
h1["a"] = "Found A"
h1[false] = "Found false"
h1["a"]
h1[false]
h1[42]
```

There are many methods defined on hashes. Useful ones include `keys` (return an array of all keys), `values` (similar for values), and `delete` (given a key, remove it and its value from the hash). Hashes also support many of the same iterators as arrays, such as `each` and `inject`, but some take the keys and the values as arguments, so consult the documentation.

A range represents a contiguous sequence of numbers (or other things, but we will focus on numbers). For example `1..100` represents the integers 1, 2, 3, ..., 100. We could use an array like `Array.new(100) {|i| i}`, but ranges are more efficiently represented and, as seen with `1..100`, there is more convenient syntax to create them. Although there are often better iterators available, a method call like `(0..n).each {|i| e}` is a lot like a for-loop from 0 to n in other programming languages.

It is worth emphasizing that duck typing lets us use ranges in many places where we might naturally expect arrays. For example, consider this method, which counts how many elements of `a` have squares less than 50:

```
def foo a
  a.count {|x| x*x < 50}
end
```

We might naturally expect `foo` to take arrays, and calls like `foo [3,5,7,9]` work as expected. But we can pass to `foo` any object with a `count` method that expects a block taking one argument. So we can also do `foo (2..10)`, which evaluates to 6.

Subclassing and Inheritance

Basic Idea and Terminology

Subclassing is an essential feature of class-based OOP. If class `C` is a subclass of `D`, then every instance of `C` is also an instance of `D`. The definition of `C` *inherits* the methods of `D`, i.e., they are part of `C`'s definition too. Moreover, `C` can *extend* by defining new methods that `C` has and `D` does not. And it can *override* methods, by changing their definition from the inherited definition. In Ruby, this is much like in Java. In Java, a subclass also inherits the field definitions of the superclass, but in Ruby fields (i.e., instance variables) are not part of a class definition because each object instance just creates its own instance variables.

Every class in Ruby except `Object` has one superclass.[4] The classes form a tree where each node is a class and the parent is its superclass. The `Object` class is the root of the tree. In class-based languages, this is

[4]Actually, the superclass of `Object` is `BasicObject` and `BasicObject` has no superclass, but this is not an important detail, so we will ignore it.

called the *class hierarchy*. By the definition of subclassing, a class has all the methods of all its ancestors in the tree (i.e., all nodes between it and the root, inclusive), subject to overriding.

Some Ruby Specifics

- A Ruby class definition specifies a superclass with `class C < D ... end` to define a new class `C` with superclass `D`. Omitting the `< D` implies `< Object`, which is what our examples so far have done.
- Ruby's built-in methods for reflection can help you explore the class hierarchy. Every object has a `class` method that returns the class of the object. Consistently, if confusingly at first, a class is itself an object in Ruby (after all, every value is an object). The class of a class is `Class`. This class defines a method `superclass` that returns the superclass.
- Every object also has methods `is_a?` and `instance_of?`. The method `is_a?` takes a class (e.g., `x.is_a? Integer`) and returns true if the receiver is an instance of `Integer` or any (transitive) subclass of `Integer`, i.e., if it is below `Integer` in the class hierarchy. The method `instance_of?` is similar but returns true only if the receiver is an instance of the class exactly, not a subclass. (Note that in Java the primitive `instanceof` is analogous to Ruby's `is_a?`.)

Using methods like `is_a?` and `instanceof` is "less object-oriented" and therefore often not preferred style. They are in conflict with duck typing.

A First Example: Point and ColorPoint

Here are definitions for simple classes that describe simple two-dimensional points and a subclass that adds a color (just represented with a string) to instances.

```
class Point
  attr_accessor :x, :y
  def initialize(x,y)
    @x = x
    @y = y
  end
  def distFromOrigin
    Math.sqrt(@x * @x  + @y * @y)
  end
  def distFromOrigin2
    Math.sqrt(x * x + y * y)
  end
end
class ColorPoint < Point
  attr_accessor :color
  def initialize(x,y,c="clear")
    super(x,y)
    @color = c
  end
end
```

There are many ways we could have defined these classes. Our design choices here include:

- We make the `@x`, `@y`, and `@color` instance variables mutable, with public getter and setter methods.
- The default "color" for a `ColorPoint` is `"clear"`.

- For pedagogical purposes revealed below, we implement the distance-to-the-origin in two different ways. The `distFromOrigin` method accesses instance variables directly whereas `distFromOrigin2` uses the getter methods on `self`. Given the definition of `Point`, both will produce the same result.

The `initialize` method in `ColorPoint` uses the `super` keyword, which allows an overriding method to call the method of the same name in the superclass. This is not required when constructing Ruby objects, but it is often desired.

Why Use Subclassing?

We now consider the style of defining colored-points using a subclass of the class `Point` as shown above. It turns out this is good OOP style in this case. Defining `ColorPoint` is good style because it allows us to reuse much of our work from `Point` and it makes sense to treat any instance of `ColorPoint` as though it "is a" `Point`.

But there are several alternatives worth exploring because subclassing is often overused in object-oriented programs, so it is worth considering at program-design time whether the alternatives are better than subclassing.

First, in Ruby, we can extend and modify classes with new methods. So we could simply change the `Point` class by replacing its `initialize` method and adding getter/setter methods for `@color`. This would be appropriate only if every `Point` object, including instances of all other subclasses of `Point`, should have a color or at least having a color would not mess up anything else in our program. Usually modifying classes is not a modular change — you should do it only if you know it will not negatively affect anything in the program using the class.

Second, we could just define `ColorPoint` "from scratch," copying over (or retyping) the code from `Point`. In a dynamically typed language, the difference in *semantics* (as opposed to style) is small: instances of `ColorPoint` will now return false if sent the message `is_a?` with argument `Point`, but otherwise they will work the same. In languages like Java/C#/C++, superclasses have effects on static typing. One advantage of *not* subclassing `Point` is that any later changes to `Point` will not affect `ColorPoint` — in general in class-based OOP, one has to worry about how changes to a class will affect any subclasses.

Third, we could have `ColorPoint` be a subclass of `Object` but have it contain an instance variable, call it `@pt`, holding an instance of `Point`. Then it would need to define all of the methods defined in `Point` to forward the message to the object in `@pt`. Here are two examples, omitting all the other methods (`x=`, `y`, `y=`, `distFromOrigin`, `distFromOrigin2`):

```
def initialize(x,y,c="clear")
  @pt = Point.new(x,y)
  @color = c
end
def x
  @pt.x # forward the message to the object in @pt
end
```

This approach is bad style since again subclassing is shorter and we want to treat a `ColorPoint` as though it "is a" `Point`. But in general, many programmers in object-oriented languages overuse subclassing. In situations where you are making a new kind of data that includes a pre-existing kind of data *as a separate sub-part of it*, this instance-variable approach is better style.

Overriding and Dynamic Dispatch

Now let's consider a different subclass of `Point`, which is for three-dimensional points:

```
class ThreeDPoint < Point
  attr_accessor :z
  def initialize(x,y,z)
    super(x,y)
    @z = z
  end
  def distFromOrigin
    d = super
    Math.sqrt(d * d + @z * @z)
  end
  def distFromOrigin2
    d = super
    Math.sqrt(d * d + z * z)
  end
end
```

Here, the code-reuse advantage is limited to inheriting methods `x`, `x=`, `y`, and `y=`, as well as using other methods in `Point` via `super`. Notice that in addition to overriding `initialize`, we used overriding for `distFromOrigin` and `distFromOrigin2`.

Computer scientists have been arguing for decades about whether this subclassing is good style. On the one hand, it does let us reuse quite a bit of code. On the other hand, one could argue that a `ThreeDPoint` is *not* conceptually a (two-dimensional) `Point`, so passing the former when some code expects the latter could be inappropriate. Others say a `ThreeDPoint` is a `Point` because you can "think of it" as its projection onto the plane where `z` equals 0. We will not resolve this legendary argument, but you should appreciate that often subclassing is bad/confusing style even if it lets you reuse some code in a superclass.

The argument against subclassing is made stronger if we have a method in `Point` like `distance` that takes another (object that behaves like a) `Point` and computes the distance between the argument and `self`. If `ThreeDPoint` wants to override this method with one that takes another (object that behaves like a) `ThreeDPoint`, then `ThreeDPoint` instances will *not* act like `Point` instances: their `distance` method will fail when passed an instance of `Point`.

We now consider a *much* more interesting subclass of `Point`. Instances of this class `PolarPoint` behave equivalently to instances of `Point` except for the arguments to `initialize`, but instances use an internal representation in terms of polar coordinates (radius and angle):

```
class PolarPoint < Point
  def initialize(r,theta)
    @r = r
    @theta = theta
  end
  def x
    @r * Math.cos(@theta)
  end
  def y
    @r * Math.sin(@theta)
  end
  def x= a
    b = y # avoids multiple calls to y method
    @theta = Math.atan(b / a)
    @r = Math.sqrt(a*a + b*b)
    self
  end
  def y= b
    a = y # avoid multiple calls to y method
    @theta = Math.atan(b / a)
    @r = Math.sqrt(a*a + b*b)
    self
  end
  def distFromOrigin
    @r
  end
  # distFromOrigin2 already works!!
end
```

Notice instances of `PolarPoint` do not have instance variables `@x` and `@y`, but the class does override the `x`, `x=`, `y`, and `y=` methods so that clients cannot tell the implementation is different (modulo round-off of floating-point numbers): they can use instances of `Point` and `PolarPoint` interchangeably. A similar example in Java would still have fields from the superclass, but would not use them. The advantage of `PolarPoint` over `Point`, which admittedly is for sake of example, is that `distFromOrigin` is simpler and more efficient.

The key point of this example is that **the subclass does not override `distFromOrigin2`, but the inherited method works correctly**. To see why, consider the definition in the superclass:

```
  def distFromOrigin2
    Math.sqrt(x * x + y * y)
  end
```

Unlike the definition of `distFromOrigin`, this method uses other method calls for the arguments to the multiplications. Recall this is just syntactic sugar for:

```
  def distFromOrigin2
    Math.sqrt(self.x() * self.x() + self.y() * self.y())
  end
```

In the superclass, this can seem like an unnecessary complication since `self.x()` is just a method that returns `@x` and methods of `Point` can access `@x` directly, as `distFromOrigin` does.

However, overriding methods `x` and `y` in a subclass of `Point` changes how `distFromOrigin2` behaves in instances of the subclass. Given a `PolarPoint` instance, its `distFromOrigin2` method is defined with the code above, but when called, `self.x` and `self.y` will call the methods defined in `PolarPoint`, not the methods defined in `Point`.

This semantics goes by many names, including *dynamic dispatch*, *late binding*, and *virtual method calls*. There is nothing quite like it in functional programming, since the way `self` is treated in the environment is special, as we discuss in more detail next.

The Precise Definition of Method Lookup

The purpose of this discussion is to consider the semantics of object-oriented language constructs, particularly calls to methods, as carefully as we have considered the semantics of functional language constructs, particularly calls to closures. As we will see, the key distinguishing feature is what `self` is bound to in the environment when a method is called. The correct definition is what we call *dynamic dispatch.*

The essential question we will build up to is given a call `e0.m(e1,e2,...en)`, what are the rules for "looking up" what method definition `m` we call, which is a non-trivial question in the presence of overriding. But first, let us notice that in general such questions about how we "look up" something are often essential to the semantics of a programming language. For example, in ML and Racket, the rules for looking up variables led to lexical scope and the proper treatment of function closures. And in Racket, we had three different forms of let-expressions exactly because they have different semantics for how to look up variables in certain subexpressions.

In Ruby, the variable-lookup rules for local variables in methods and blocks are not too different from in ML and Racket despite some strangeness from variables not being declared before they are used. But we also have to consider how to "look up" instance variables, class variables, and methods. In all cases, the answer depends on the object bound to `self` — and `self` is treated specially.

In any environment, `self` maps to some object, which we think of as the "current object" — the object currently executing a method. To look up an instance variable `@x`, we use the object bound to `self` – each object has its own state and we use `self`'s state. To look up a class variable `@@x`, we just use the state of the object bound to `self.class` instead. To look up a method `m` for a method call is more sophisticated...

In class-based object-oriented languages like Ruby, the rule for evaluating a method call like `e0.m(e1,...,en)` is:

- Evaluate `e0`, `e1`, ..., `en` to values, i.e., objects `obj0`, `obj1`, ..., `objn`.
- Get the class of `obj0`. Every object "knows its class" at run-time. Think of the class as part of the state of `obj0`.
- Suppose `obj0` has class `A`. If `m` is defined in `A`, call that method. Otherwise recur with the superclass of `A` to see if it defines `m`. Raise a "method missing" error if neither `A` nor any of its superclasses define `m`. (Actually, in Ruby the rule is actually to instead call a method called `method_missing`, which any class can define, so we again start looking in `A` and then its superclass. But most classes do not define `method_missing` and the definition of it in `Object` raises the error we expect.)
- We have now found the method to call. If the method has *formal arguments* (i.e., argument names or parameters) `x1`, `x2`, ..., `xn`, then the environment for evaluating the body will map `x1` to `obj1`, `x2` to `obj2`, etc. But there is one more thing that is the essence of object-oriented programming and has no real analogue in functional programming: We always have `self` in the environment. **While evaluating the method body, `self` is bound to `obj0`, the object that is the "receiver" of the message.**

The binding of `self` in the callee as described above is what is meant by the synonyms "late-binding," "dynamic dispatch," and "virtual method calls." It is central to the semantics of Ruby and other OOP languages. It means that when the body of `m` calls a method on `self` (e.g., `self.someMethod 34` or just `someMethod 34`), we use the class of `obj0` to resolve `someMethod`, *not necessarily* the class of the method we are executing. This is why the `PolarPoint` class described above works as it does.

There are several important comments to make about this semantics:

- Ruby's mixins complicate the lookup rules a bit more, so the rules above are actually simplified by ignoring mixins. When we study mixins, we will revise the method-lookup semantics accordingly.
- This semantics is quite a bit more complicated than ML/Racket function calls. It may not seem that way if you learned it first, which is common because OOP and dynamic dispatch seem to be a focus in many introductory programming courses. But it is truly more complicated: we have to treat the notion of `self` differently from everything else in the language. Complicated does not necessarily mean it is inferior or superior; it just means the language definition has more details that need to be described. This semantics has clearly proved useful to many people.
- *Optional:* Java and C# have significantly more complicated method-lookup rules. They do have dynamic dispatch as described here, so studying Ruby should help understand the semantics of method lookup in those languages. But they *also* have *static overloading*, in which classes can have multiple methods with the same name but taking different types (or numbers) of arguments. So we need to not just find *some* method with the right name, but we have to find one that *matches* the types of the arguments at the call. Moreover, multiple methods might match and the language specifications have a long list of complicated rules for finding the *best* match (or giving a type error if there is no best match). In these languages, one method overrides another only if its arguments have the same type and number. None of this comes up in Ruby where "same method name" always means overriding and we have no static type system. In C++, there are even more possibilities: we have static overloading and different forms of methods that either do or do not support dynamic dispatch.

Dynamic Dispatch Versus Closures

To understand how dynamic dispatch differs from the lexical scope we used for function calls, consider this simple ML code that defines two mutually recursive functions:

```
fun even x = if x=0 then true else odd (x-1)
and odd  x = if x=0 then false else even (x-1)
```

This creates two closures that both have the other closure in their environment. If we later shadow the `even` closure with something else, e.g.,

```
fun even x = false
```

that will *not* change how `odd` behaves. When `odd` looks up `even` in the environment where `odd` was defined, it will get the function on the first line above. That is "good" for understanding how `odd` works *just from looking where is defined.* On the other hand, suppose we wrote a better version of `even` like:

```
fun even x = (x mod 2) = 0
```

Now our `odd` is not "benefiting from" this optimized implementation.

In OOP, we can use (abuse?) subclassing, overriding, and dynamic dispatch to change the behavior of `odd` by overriding `even`:

```
class A
  def even x
     if x==0 then true else odd(x-1) end
  end
  def odd x
     if x==0 then false else even(x-1) end
  end
end
class B < A
  def even x # changes B's odd too!
     x % 2 == 0
  end
end
```

Now (`B.new.odd 17`) will execute faster because `odd`'s call to `even` will resolve to the method in `B` – all because of what `self` is bound to in the environment. While this is certainly convenient in the short example above, it has real drawbacks. We cannot look at one class (`A`) and know how calls to the code there will behave. In a subclass, what if someone overrode `even` and did not know that it would change the behavior of `odd`? Basically, any calls to methods that might be overridden need to be thought about very carefully. It is likely often better to have private methods that cannot be overridden to avoid problems. Yet overriding and dynamic dispatch is the biggest thing that distinguishes object-oriented programming from functional programming.

Optional: Implementing Dynamic Dispatch Manually in Racket

Although this topic is optional, we highly recommend it because often a great way to understand something is to see it implemented in terms of something else.

Let's now consider *coding up* objects and dynamic dispatch in Racket using nothing more than pairs and functions.[5] This serves two purposes:

- It demonstrates that one language's *semantics* (how the primitives like message send work in the language) can typically be coded up as an *idiom* (simulating the same behavior via some helper functions) in another language. This can help you be a better programmer in different languages that may not have the features you are used to.
- It gives a lower-level way to understand how dynamic dispatch "works" by seeing how we would do it manually in another language. An interpreter for an object-oriented language would have to do something similar for automatically evaluating programs in the language.

Also notice that we did an analogous exercise to better understand closures earlier in the course: We showed how to get the effect of closures in Java using objects and interfaces or in C using function pointers and explicit environments.

Our approach will be different from what Ruby (or Java for that matter) actually does in these ways:

- Our objects will just contain a list of fields and a list of methods. This is not "class-based," in which an object would have a list of fields and a class-name and then the class would have the list of methods. We could have done it that way instead.
- Real implementations are more efficient. They use better data structures (based on arrays or hashtables) for the fields and methods rather than simple association lists.

[5]Though we did not study it, Racket has classes and objects, so you would not actually want to do this in Racket. The point is to understand dynamic dispatch by manually coding up the same idea.

Nonetheless, the key ideas behind how you implement dynamic dispatch still come through. By the way, we are wise to do this in Racket rather than ML, where the types would get in our way. In ML, we would likely end up using "one big datatype" to give all objects and all their fields the same type, which is basically awkwardly programming in a Racket-like way in ML. (Conversely, typed OOP languages are often no friendlier to ML-style programming unless they add separate constructs for generic types and closures.)

Our objects will just have fields and methods:

```
(struct obj (fields methods))
```

We will have `fields` hold an immutable list of *mutable* pairs where each element pair is a symbol (the field name) and a value (the current field contents). With that, we can define helper functions `get` and `set` that given an object and a field-name, return or mutate the field appropriately. Notice these are just plain Racket functions, with no special features or language additions. We do need to define our own function, called `assoc-m` below, because Racket's `assoc` expects an immutable list of immutable pairs.

```
(define (assoc-m v xs)
  (cond [(null? xs) #f]
        [(equal? v (mcar (car xs))) (car xs)]
        [#t (assoc-m v (cdr xs))]))

(define (get obj fld)
  (let ([pr (assoc-m fld (obj-fields obj))])
    (if pr
        (mcdr pr)
        (error "field not found"))))

(define (set obj fld v)
  (let ([pr (assoc-m fld (obj-fields obj))])
    (if pr
        (set-mcdr! pr v)
        (error "field not found"))))
```

More interesting is calling a method. The `methods` field will also be an association list mapping method names to functions (no mutation needed since we will be less dynamic than Ruby). The key to getting dynamic dispatch to work is that these functions will all take an extra *explicit* argument that is *implicit* in languages with built-in support for dynamic dispatch. This argument will be "self" and our Racket helper function for sending a message will simply pass in the correct object:

```
(define (send obj msg . args)
  (let ([pr (assoc msg (obj-methods obj))])
    (if pr
        ((cdr pr) obj args)
        (error "method not found" msg))))
```

Notice how the function we use for the method gets passed the "whole" object `obj`, which will be used for any sends to the object bound to `self`. (The code above uses Racket's support for variable-argument functions because it is convenient — we could have avoided it if necessary. Here, `send` can take any number of arguments greater than or equal to 2. The first argument is bound to `obj`, the second to `msg`, and all others are put in a list (in order) that is bound to `args`. Hence we expect `(cdr pr)` to be a function that takes two arguments: we pass `obj` for the first argument and the list `args` for the second argument.)

Now we can define `make-point`, which is just a Racket function that produces a point object:

```
(define (make-point _x _y)
  (obj
   (list (mcons 'x _x)
         (mcons 'y _y))
   (list (cons 'get-x (lambda (self args) (get self 'x)))
         (cons 'get-y (lambda (self args) (get self 'y)))
         (cons 'set-x (lambda (self args) (set self 'x (car args))))
         (cons 'set-y (lambda (self args) (set self 'y (car args))))
         (cons 'distToOrigin
               (lambda (self args)
                 (let ([a (send self 'get-x)]
                       [b (send self 'get-y)])
                   (sqrt (+ (* a a) (* b b)))))))))
```

Notice how each of the methods takes a first argument, which we just happen to call `self`, which has no special meaning here in Racket. We then use `self` as an argument to `get`, `set`, and `send`. If we had some other object we wanted to send a message to or access a field of, we would just pass that object to our helper functions by putting it in the `args` list. In general, the second argument to each function is a list of the "real arguments" in our object-oriented thinking.

By using the `get`, `set`, and `send` functions we defined, making and using points "feels" just like OOP:

```
(define p1 (make-point 4 0))
(send p1 'get-x)        ; 4
(send p1 'get-y)        ; 0
(send p1 'distToOrigin) ; 4
(send p1 'set-y 3)
(send p1 'distToOrigin) ; 5
```

Now let's simulate subclassing...

Our encoding of objects does not use classes, but we can still create something that reuses the code used to define points. Here is code to create points with a color field and getter/setter methods for this field. The key idea is to have the constructor create a point object with `make-point` and then extend this object by creating a new object that has the extra field and methods:

```
(define (make-color-point _x _y _c)
  (let ([pt (make-point _x _y)])
    (obj
     (cons (mcons 'color _c)
           (obj-fields pt))
     (append (list
              (cons 'get-color (lambda (self args) (get self 'color)))
              (cons 'set-color (lambda (self args) (set self 'color (car args)))))
            (obj-methods pt)))))
```

We can use "objects" returned from `make-color-point` just like we use "objects" returned from `make-point`, plus we can use the field `color` and the methods `get-color` and `set-color`.

The essential distinguishing feature of OOP is dynamic dispatch. Our encoding of objects "gets dynamic dispatch right" but our examples do not yet demonstrate it. To do so, we need a "method" in a "superclass" to call a method that is defined/overridden by a "subclass." As we did in Ruby, let's define polar points by adding new fields and overriding the `get-x`, `get-y`, `set-x`, and `set-y` methods. A few details about the code below:

- As with color-points, our "constructor" uses the "superclass" constructor.
- As would happen in Java, our polar-point objects still have `x` and `y` fields, but we never use them.
- For simplicity, we just override methods by putting the replacements earlier in the method list than the overridden methods. This works because `assoc` returns the first matching pair in the list.

Most importantly, the `distToOrigin` "method" still works for a polar point because the method calls in its body will use the procedures listed with `'get-x` and `'get-y` in the definition of `make-polar-point` just like dynamic dispatch requires. The correct behavior results from our `send` function passing the whole object as the first argument.

```
(define (make-polar-point _r _th)
  (let ([pt (make-point #f #f)])
    (obj
     (append (list (mcons 'r _r)
                   (mcons 'theta _th))
             (obj-fields pt))
     (append
      (list
       (cons 'set-r-theta
             (lambda (self args)
               (begin
                 (set self 'r (car args))
                 (set self 'theta (cadr args)))))
       (cons 'get-x (lambda (self args)
                      (let ([r (get self 'r)]
                            [theta (get self 'theta)])
                        (* r (cos theta)))))
       (cons 'get-y (lambda (self args)
                      (let ([r (get self 'r)]
                            [theta (get self 'theta)])
                        (* r (sin theta)))))
       (cons 'set-x (lambda (self args)
                      (let* ([a     (car args)]
                             [b     (send self 'get-y)]
                             [theta (atan (/ b a))]
                             [r     (sqrt (+ (* a a) (* b b)))])
                        (send self 'set-r-theta r theta))))
       (cons 'set-y (lambda (self args)
                      (let* ([b     (car args)]
                             [a     (send self 'get-x)]
                             [theta (atan (/ b a))]
                             [r     (sqrt (+ (* a a) (* b b)))])
                        (send self 'set-r-theta r theta)))))
      (obj-methods pt)))))
```

We can create a polar-point object and send it some messages like this:

```
(define p3 (make-polar-point 4 3.1415926535))
(send p3 'get-x) ; 4
(send p3 'get-y) ; 0 (or a slight rounding error)
(send p3 'distToOrigin) ; 4 (or a slight rounding error)
(send p3 'set-y 3)
(send p3 'distToOrigin) ; 5 (or a slight rounding error)
```

Programming Languages (Coursera / University of Washington) Assignment 6

Note: Playing Tetris is fun, but playing too much can make the assignment take longer than necessary.

Overview

This assignment is about a Tetris game written in Ruby. There are four Ruby files involved:

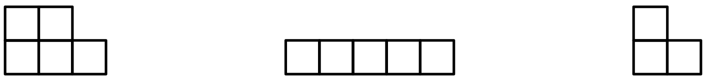
1. The Ruby code in `hw6provided.rb` implements a simple but fully functioning Tetris game (see, for example, `http://en.wikipedia.org/wiki/Tetris#Gameplay`).
2. In `hw6assignment.rb`, you will create a second game that is Tetris with some *enhancements*, described below. This is the only file you will submit (other than, as usual, a file you used for testing).
3. The Ruby code in `hw6runner.rb` is the main starting point. From the command-line (**not** within `irb`), you can run `ruby hw6runner.rb` to play a Tetris game that includes your enhancements. To use the original game rather than your code in `hw6assignment.rb`, run `ruby hw6runner.rb original`.
4. The Ruby code in `hw6graphics.rb` provides a simple graphics library, tailored to Tetris. It is used by the Tetris game in `hw6provided.rb`. Your code can also use the classes and methods in `hw6graphics.rb` (as well as the classes and methods in `hw6provided.rb`), except for the methods marked with comments as not to be called by student code. Your code *cannot* use the Tk graphics library directly.

You will turn in only `hw6assignment.rb`, so your solution cannot involve modifying any of the provided code. Instead, use subclassing (since, after all, this is the OOP portion of the course) to nonetheless *reuse* as much of the code as possible. Some code-copying will still be necessary, but you should try to minimize it. Your solution should also *not* change any of the provided classes in any way. You will define subclasses that behave differently. As a result, the provided Tetris game will run without change.

Much of the work in this assignment is understanding the provided code. There are parts you will need to understand very well and other parts you will not need to understand. Part of the challenge is figuring out which parts are which. This experience is fairly realistic.

Enhancements

1. In your game, the player can press the 'u' key to make the piece that is falling rotate 180 degrees. (Note it is normal for this to make some pieces appear to move slightly.)
2. In your game, instead of the pieces being randomly (and uniformly) chosen from the 7 classic pieces, the pieces are randomly (and uniformly) chosen from 10 pieces. They are the classic 7 and these 3:

   The initial rotation for each piece is also chosen randomly.
3. In your game, the player can press the '`c`' key to *cheat*: If the score is less than 100, nothing happens. Else the player loses 100 points (cheating costs you) and the next piece that appears will be:

   The piece after is again chosen randomly from the 10 above (unless, of course, the player hits '`c`' while the "cheat piece" is falling and still has a large enough score). Hitting 'c' multiple times while a single piece is falling should behave no differently than hitting it once.

How To Run The Game

We recommend you *not* use `irb` to load `hw6runner.rb` and start a game. We have not had success getting the graphics library to restart properly more than once inside `irb`, so you would likely have to exit the REPL after playing each game anyway. You can use the REPL for testing individual methods and exploring the program, but to launch a game, go outside `irb` and run `ruby hw6runner.rb` (or to make sure the original unenhanced game still works correctly, run `ruby hw6runner.rb original`). Make sure the 4 Ruby files are in the same directory and run the `ruby` command (or `irb` when exploring) from that directory.

Requirements

For our testing scripts to work, you **must** follow these guidelines.

- Your game should have all the features of the original Tetris game, as well as the enhancements.
- The subclasses you create must start with `My` followed by the name of the original class. For example, your Tetris class should be called `MyTetris`. We have provided empty class definitions for you to complete. You do not need any other classes.
- Do not add to or modify any classes defined in other files or the standard library.
- It is required that your board `MyBoard` has a `next_piece` method that provides the same functionality that `Board`'s `next_piece` provides, which is that it sets @`current_block` to the next piece that will fall, which might or might not be the cheat piece.
- You must have a `MyPiece` class and it must define a class constant `All_My_Pieces` that contains exactly the ten "normal" pieces (i.e., the initial seven plus the three additional pieces from enhancement two). It must not contain your cheat piece. It must be in the same format as the `All_Pieces` array in the provided code.
- All your new pieces, including your cheat piece, must use the same format as the provided pieces. Hint: Be particularly careful to have enough nesting in your arrays or your game may be subtly, *almost* imperceptibly, incorrect.
- **Do not use the Tk library directly in any way.** The only use of Tk should occur indirectly by using instances of classes defined in `hw6graphics.rb` as needed (only a little is needed). Do not have `require 'tk'` in your `hw6assignment.rb` file (or any other use of `require` for that matter).

Advice and Guidance

- For the piece you are adding that has 5 squares in it and is not a line, be sure to add the piece as pictured and not its mirror image.
- It takes time to understand code you are given. Be patient and work methodically. You might try changing things to see what happens, but be sure you undo any changes you make to the provided code.
- The sample solution is about 85 lines of code. As always, that is just a rough guideline; your solution might be longer or shorter.
- While you should not copy code unless necessary, some copying will be necessary. After all, the original game may not have functionality broken down into overridable methods the way you would want and you are not allowed to change the provided code. This is a fairly realistic situation.

(Lack Of) Challenge Problem

The best thing to do to continue the assignment is to implement an additional enhancement (or more) to your game, but we do not have a reasonable way to grade this and additional enhancements could possibly mess up auto-grading. So:

- This assignment will not have a graded challenge problem.
- Be sure to turn in a solution that adds *only* the three asked-for enhancements.
- Then enjoy trying additional enhancements. Be creative.

We encourage you to share on the discussion forum *what you did* but **not** *how you did it* as that would take away the fun from others who might try to repeat your accomplishment.

Turn-in Instructions (same as usual except as described on the website to support multiple Ruby versions): First, follow the instructions on the course website to submit your solution file (not your testing file) for auto-grading. Do not proceed to the peer-assessment submission until you receive a high-enough grade from the auto-grader: Doing peer assessment requires instructions that include a sample solution, so these instructions will be "locked" until you receive high-enough auto-grader score. Then submit your same solution file again for peer assessment and follow the peer-assessment instructions.

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 1
Extending the Code With New Operations or Variants 4
Binary Methods With Functional Decomposition 6
Binary Methods in OOP: Double Dispatch 7
Optional: Multimethods 11
Multiple Inheritance 12
Mixins 13
Java/C#-Style Interfaces 16
Optional: Abstract Methods 16

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.

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

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.

Programming Languages (Coursera / University of Washington) Assignment 7

Set-up: For this assignment, you will complete and extend two implementations of an interpreter for a small "language" for two-dimensional geometry objects. An implementation in SML is mostly completed for you. An implementation in Ruby is mostly not completed. The SML implementation is structured with functions and pattern-matching. The Ruby implemented is structured with subclasses and methods, including some mind-bending double dispatch and other dynamic dispatch to stick with an OOP style even where your instructor thinks the functional style is easier to understand.

Download and edit `hw7.sml` and `hw7.rb` from the course website. Some example tests are also provided.

Language Semantics:

Our "language" has five kinds of values and four other kinds of expressions. The representation of expressions depends on the metalanguage (SML or Ruby), with this same semantics:

- A `NoPoints` represents the empty set of two-dimensional points.
- A `Point` represents a two-dimensional point with an x-coordinate and a y-coordinate. Both coordinates are floating-point numbers.
- A `Line` is a non-vertical infinite line in the plane, represented by a *slope* and an *intercept* (as in $y = mx + b$ where m is the slope and b is the intercept), both floating-point numbers.
- A `VerticalLine` is an infinite vertical line in the plane, represented by its x-coordinate.
- A `LineSegment` is a (finite) line segment, represented by the x- and y-coordinates of its endpoints (so four total floating-point numbers).
- An `Intersect` expression is not a value. It has two subexpressions. The semantics is to evaluate the subexpressions (in the same environment) and then return the value that is the intersection (in the geometric sense) of the two subresults. For example, the intersection of two lines could be one of:
  - `NoPoints`, if the lines are parallel
  - a `Point`, if the lines intersect
  - a `Line`, if the lines have the same slope and intercept (see the note below about what "the same" means for floating-point numbers)
- A `Let` expression is not a value. It is like let-expressions in other languages we have studied: The first subexpression is evaluated and the result bound to a variable that is added to the environment for evaluating the second subexpression.
- A `Var` expression is not a value. It is for using variables in the environment: We look up a string in the environment to get a geometric value.
- A `Shift` expression is not a value. It has a *deltaX* (a floating-point number), a *deltaY* (a floating-point number), and a subexpression. The semantics is to evaluate the subexpression and then *shift* the result by `deltaX` (in the x-direction; positive is "to the right") and `deltaY` (in the y-direction; positive is "up"). More specifically, shifting for each form of value is as follows:
  - `NoPoints` remains `NoPoints`.
  - A `Point` representing (x, y) becomes a `Point` representing $(x + \mathit{deltaX}, y + \mathit{deltaY})$.
  - A `Line` with slope m and intercept b becomes a `Line` with slope m and an intercept of $b + \mathit{deltaY} - m \cdot \mathit{deltaX}$.
  - A `VerticalLine` becomes a `VerticalLine` shifted by *deltaX*; the *deltaY* is irrelevant.
  - A `LineSegment` has its endpoints shift by *deltaX* and *deltaY*.

Note on Floating-Point Numbers:

Because arithmetic with floating-point numbers can introduce small rounding errors, it is rarely appropriate to use equality to decide if two floating-point numbers are "the same." Instead, the provided code uses a helper function/method to decide if two floating-point numbers are "real close" (for our purposes, within .00001) and all your code should follow this approach as needed. For example, two points are the same if their x-coordinates are within .00001 and their y-coordinates are within .00001.

Expression Preprocessing:

To simplify the interpreter, we first preprocess expressions. Preprocessing takes an expression and produces a new, equivalent expression with the following invariants:

- No `LineSegment` anywhere in the expression has endpoints that are the same as (i.e., real close to) each other. Such a line-segment should be replaced with the appropriate `Point`. For example in ML syntax, `LineSegment(3.2,4.1,3.2,4.1)` should be replaced with `Point(3.2,4.1)`.
- Every `LineSegment` has its first endpoint (the first two `real` values in SML) to the left (lower x-value) of the second endpoint. If the x-coordinates of the two endpoints are the same (real close), then the `LineSegment` has its first endpoint below (lower y-value) the second endpoint. For any `LineSegment` not meeting this requirement, replace it with a `LineSegment` with the same endpoints reordered.

The SML Code:

Most of the SML solution is given to you. All you have to do is add preprocessing (problem 1) and `Shift` expressions (problem 2). The sample solution added much less than 50 lines of code. As always, line counts are just a rough guide.

Notice the SML code is organized around a datatype-definition for expressions, functions for the different operations, and pattern-matching to identify different cases. The interpreter `eval_prog` uses a helper function `intersect` with cases for every combination of geometric value (so with 5 kinds of values there are 25 cases though some are handled together via pattern-matching). The surprisingly complicated part is the algorithm for intersecting two line segments.

The Ruby Code:

Much of the Ruby solution is not given to you. To get you started in the desired way, we have defined classes for each kind of expression in our language, as well as appropriate superclasses. We have implemented parts of each class and left comments with what you need to do to complete the implementation as described in more detail in problems 3 and 4. The sample solution added about 200 lines of Ruby code, many of which were `end`. As always, line counts are just a rough guide.

Notice the Ruby code is organized around classes where each class has methods for various operations. All kinds of expressions need methods for preprocessing and evaluation. They are subclasses of `GeometryExpression` just like all ML constructors are part of the `geom_exp` datatype (though the `GeometryExpression` class turns out not to be so useful). The value subclasses also need methods for shifting and intersection and they subclass `GeometryValue` so that some shared methods can be inherited (in analogy with some uses of wildcard patterns and helper functions in ML).

Your Ruby code should follow these general guidelines:

- All your geometry-expression objects should be *immutable*: assign to instance variables only when constructing an object. To "change" a field, create a new object.
- The geometry-expression objects have public getter methods: like in the SML code, the entire program can assume the expressions have various coordinates, subexpressions, etc.
- Unlike in SML, you do not need to define exceptions since without a type-checker we can just "assume" the right objects are used in the right places. You can also use `raise` with just a string as appropriate.

- Follow OOP-style. In particular, operations should be instance methods and you should **not use methods like `is_a?`, `instance_of?`, `class`, etc.** This makes problem 4 much more difficult, which is the purpose of the problem.

Advice for Approaching the Assignment:

- Understand the high-level structure of the code and how the SML and Ruby files are structured in different ways before diving into the details.
- **Approach the questions in order even though there is some flexibility (e.g., it is possible to do the Ruby problems before the SML problems).**
- Because almost all the SML code is given to you, for much of the Ruby implementation, you can port the corresponding part of the SML solution. Doing so makes your job **much** easier (e.g., you need not re-figure out facts about geometry). Porting existing code to a new language is a useful and realistic skill to develop. It also helps teach the similarities and differences between languages.
- Be sure to test each line of your Ruby code. Dynamically typed languages require testing things that other languages catch for you statically. Ruby will not even tell you statically if you misspell a method name or instance variable.

The Problems (Finally):

1. Implement an SML function `preprocess_prog` of type `geom_exp -> geom_exp` to implement expression preprocessing as defined above. The idea is that evaluating program `e` would be done with `eval_prog (preprocess_prog e, [])` where the `[]` is the empty list for the empty environment.
2. Add shift expressions as defined above to the SML implementation by adding the constructor `Shift of real * real * geom_exp` to the definition of `geom_exp` and adding appropriate branches to `eval_prog` and `preprocess_prog`. (The first `real` is *deltaX* and the second is *deltaY*.) Do *not* change other functions. In particular, there is no need to change `intersect` because this function is used only for values in our geometry language and shift expressions are not geometry values.
3. Complete the Ruby implementation *except for intersection*, which means skip for now additions to the `Intersect` class and, more importantly, methods related to intersection in other classes. Do not modify the code given to you. Follow this approach:
   - Every subclass of `GeometryExpression` should have a `preprocess_prog` method that takes no arguments and returns the geometry object that is the result of preprocessing `self`. To avoid mutation, return a new instance of the same class unless it is trivial to determine that `self` is already an appropriate result.
   - Every subclass of `GeometryExpression` should have an `eval_prog` method that takes one argument, the environment, which you should represent as an array whose elements are two-element arrays: a Ruby string (the variable name) in index 0 and an object that is a value in our language in index 1. As in any interpreter, pass the appropriate environment when evaluating subexpressions. (This is fairly easy since we do not have closures.) To make sure you handle both scope and shadowing correctly:
     - Do not ever mutate an environment; create a new environment as needed instead. Be careful what methods you use on arrays to avoid mutation.
     - The `eval_prog` method in `Var` is given to you. Make sure the environments you create work correctly with this definition.

   The result of `eval_prog` is the result of "evaluating the expression represented by `self`," so, as we expect with OOP style, the cases of ML's `eval_prog` are spread among our classes, just like with `preprocess_prog`.

- Every subclass of `GeometryValue` should have a `shift` method that takes two arguments `dx` and `dy` and returns the result of shifting `self` by `dx` and `dy`. In other words, all values in the language "know how to shift themselves to create new objects." Hence the `eval_prog` method in the `Shift` class should be very short.
- Remember you should not use any method like `is_a?`, `instance_of?`, `class`, etc.
- Analogous to SML, an overall program `e` would be evaluated via `e.preprocess_prog.eval_prog []` (notice we use an array for the environment).

4. Implement intersection in your Ruby solution following the directions here, in which we require both double dispatch and a separate use of dynamic dispatch for the line-segment case. Remember all the different cases in ML will appear somewhere in the Ruby solution, just arranged very differently.

   - Implement `preprocess_prog` and `eval_prog` in the `Intersect` class. This is not difficult, much like your prior work in the `Shift` class is not difficult. This is because every subclass of `GeometryValue` will have an `intersect` method that "knows how to intersect itself" with another geometry-value passed as an argument.
   - Every subclass of `GeometryValue` needs an `intersect` method, but these will be short. The argument is another geometry-value, but we do not know what kind. So we use double dispatch and call the appropriate method on the argument passing `self` to the method. For example, the `Point` class has an `intersect` method that calls `intersectPoint` with `self`.
   - So methods `intersectNoPoints`, `intersectPoint`, `intersectLine`, `intersectVerticalLine`, and `intersectLineSegment` defined in each of our 5 subclasses of `GeometryValue` handle the 25 possible intersection combinations:
     - The 9 cases involving `NoPoints` are done for you. See the `GeometryValue` class — there is nothing more you need to do.
     - Next do the 9 remaining cases involving combinations that do not involve `LineSegment`. You will need to understand double-dispatch to avoid `is_a?` and `instance_of?`. As in the ML code, 3 of these 9 cases can just use one of the other cases because intersection is commutative.
     - What remains are the 7 cases where one value is a `LineSegment` and the other is not `NoPoints`. These cases are all "done" for you because all subclasses of `GeometryValue` inherit an `intersectLineSegment` method that will be correct for all of them. But it calls `intersectWithSegmentAsLineResult`, which you need to implement for each subclass of `GeometryValue`. Here is how this method should work:
       - It takes one argument, which is a line segment. (In ML the corresponding variable was a `real*real*real*real`, but here it will actually be an instance of `LineSegment` and you can use the getter methods `x1`, `y1`, `x2`, and `y2` as needed.)
       - It *assumes* that `self` is the intersection of (1) some not-provided geometry-value and (2) the line (vertical or not) containing the segment given as an argument.
       - It *returns* the intersection of the not-provided geometry-value and the segment given as an argument.

       Together the 5 `intersectWithSegmentAsLineResult` methods you write will implement the same algorithm as on lines 110–169 of the ML code.

5. **(Lack Of) Challenge Problem:** As in the previous homework, the most educational challenge problem is not something we can reasonably auto-grade or peer asssess, so we encourage doing it even though it will not count toward your grade: Make a third version of your solution in a statically typed OOP language like Java or C#. Follow the structure of your Ruby solution, with no use of type casts or features like Java's `instanceof`. You will need to have abstract methods and abstract classes that you then subclass. (Naturally, you can also enjoy the challenge of implementing your solution in any other programming language you like.)

Turn-in Instructions:

Because we are using two different languages, the turn-in instructions are slightly different than for other assignments. Follow the instructions on the course website to turn in your files as follows:

- **For auto-grading, you will turn in two files**: Your ML solution and your Ruby solution. This is done on the website by having two assignment "parts."
- **For peer assessment, you will turn in only your Ruby code**, using the separate peer-assessment system, as for previous assignments. We will not use peer assessment for your ML code.

As always, do not proceed to the peer-assessment submission until you receive a high-enough grade from the auto-grader: Doing peer assessment requires instructions that include a sample solution, so these instructions will be "locked" until you receive high-enough auto-grader score. Then submit your file for peer assessment and follow the peer-assessment instructions.

Coursera Programming Languages Course
Section 10 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

Introduction to Subtyping 1

A Made-Up Language of Records 2

Wanting Subtyping 3

The Subtyping Relation 3

Depth Subtyping: A Bad Idea With Mutation 4

Optional: The Problem With Java/C# Array Subtyping 5

Function Subtyping 7

Subtyping for OOP 9

Optional: Covariant `self/this` 10

Generics Versus Subtyping 10

Bounded Polymorphism 13

Optional: Additional Java-Specific Bounded Polymorphism 14

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.

Optional: 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.

Optional: 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.

Optional: 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.[1] 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

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

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
        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`.