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

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 (or a slight rounding error)
(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)
```