CHAPTER 5

Functional Programming Techniques

It doesn't take much time to realize that Ruby is a deeply object-oriented language that openly steals from the best of Smalltalk. But Matz is an equal-opportunity thief, and he has snatched features from various other languages as well, including Lisp. This means that although Ruby has its roots in object-oriented principles, we also have some of the high-level constructs that facilitate functional programming techniques.

Rightfully speaking, Ruby is not a functional programming language. Though it is possible to come close with great effort, Ruby simply lacks a number of the key aspects of functional languages. Virtually all state in Ruby is mutable, and because it is an object-oriented language, we tend to focus on state management rather than elimination of state. Ruby lacks tail call optimization,* making recursion highly inefficient. Beyond these key things, there are plenty of other subtleties that purists can discuss at length if you let them.

However, if you let Ruby be Ruby, you can benefit from certain functional techniques while still writing object-oriented code. This chapter will walk you through several practices that are inspired by functional languages and that have practical value for solving problems in Ruby. We'll look at things like lazy evaluation, memoization, infinite lists, higher-order procedures, and other stuff that has a nice academic ring to it. Along the way, I'll do my best to show you that this stuff isn't simply abstract and mathematical, but can meaningfully be used in day-to-day code.

Let's start by taking a look at how we're already using lazy evaluation whether we realize it or not, and then walk through a popular Ruby library that can simplify this for us. As in other chapters, we'll take a peek under the hood to see what's really going on.

* This can be enabled in YARV at compile time, but it is still experimental.

Laziness Can Be a Virtue (A Look at lazy.rb)

If you've been writing Ruby for any amount of time, you've probably already written some code that makes use of lazy evaluation. Before we go further, let's look at a couple examples of lazy code that you will likely recognize on sight, if not by name.

For starters, `Proc` objects are, by definition, lazy:

```
a = lambda { File.read("foo.txt") }
```

When you create a `Proc` object using `lambda`, the code is not actually executed until you call the block. Therefore, a `Proc` is essentially a chunk of code that gets executed on demand, rather than in place. So if we actually wanted to read the contents of this file, we'd need to do this:

```
b = a.call
```

In essence, code is said to be evaluated lazily if it is executed only at the time it is actually needed, not at the time it was defined. However, this behavior is not necessarily limited to blocks; we can do this with populating data for our objects as well. Let's take a look at a simplified model of Prawn's table cells for a more complete example:

```
class Cell

  FONT_HEIGHT = 10
  FONT_WIDTH  = 8

  def initialize(text)
    @text = text
  end

  attr_accessor :text
  attr_writer :width, :height

  def width
    @width ||= calculate_width
  end

  def height
    @height ||= calculate_height
  end

  def to_s
    "Cell(#{width}x#{height})"
  end

  private

  def calculate_height
    @text.lines.count * FONT_HEIGHT
  end

  def calculate_width
    @text.lines.map { |e| e.length }.max * FONT_WIDTH
```

```
    end

  end
```

In this example, `Cell#width` and `Cell#height` can be calculated based on the text in the cell, or they can be manually set. Because we don't need to know the exact dimensions of a cell until we render it, this is a perfect case for lazy evaluation. Even though the calculations may not be expensive on their own, they add up when dealing with thousands or hundreds of thousands of cells. Luckily, it's easy to avoid any unnecessary work.

By just looking at the core bits of this object, we can get a clearer sense of what's going on:

```
class Cell
  attr_writer :width, :height

  def width
    @width ||= calculate_width
  end

  def height
    @height ||= calculate_height
  end
end
```

It should now be plain to see what's happening. If `@width` or `@height` have already been set, their calculations are never run. We also can see that these calculations will in the worst case be run exactly once, storing the return value as needed.

The idea here is that now we won't have to worry about calculating dimensions of preset `Cell` objects, and that those that calculate their dimensions will not need to repeat that calculation each time they are used. I am hoping that readers are familiar with this Ruby idiom already, but if you are struggling with it a little bit, just stop for a moment and toy with this in *irb* and it should quickly become clear how things work:

```
>> cell = Cell.new("Chunky Bacon\nIs Very\nDelicious")
>> cell.width = 1000
=> 1000
>> cell.to_s
=> "Cell(1000x30)"
>> cell.height = 500
=> 500
>> cell.to_s
=> "Cell(1000x500)"
```

Though this process is relatively straightforward, we can probably make it better. It's sort of annoying to have to build special accessors for our `@width` and `@height` when what we're ultimately doing is setting default values for them, which is normally something we do in our constructor. What's more, our solution feels a little primitive in nature, at least aesthetically.

This is where MenTaLguY's *lazy.rb* comes in. It provides a method called `promise()` that does exactly what we want, in a much nicer way. The following code can be used to replace our original implementation:

```
require "lazy"

class Cell

  FONT_HEIGHT = 10
  FONT_WIDTH  = 8

  def initialize(text)
    @text   = text
    @width  = promise { calculate_width }
    @height = promise { calculate_height }
  end

  attr_accessor :text, :width, :height

  def to_s
    "Cell(#{width}x#{height})"
  end

  private

  def calculate_height
    @text.lines.count * FONT_HEIGHT
  end

  def calculate_width
    @text.lines.map { |e| e.length }.max * FONT_WIDTH
  end

end
```

Gone are our special accessors, and in their place, we have simple `promise()` calls in our constructor. This method returns a simple proxy object that wraps a block of code that is designed to be executed later. Once you call any methods on this object, it passes them along to whatever your block evaluates to. If this is tough to get your head around, again *irb* is your friend:

```
>> a = promise { 1 }
=> #<Lazy::Promise computation=#<Proc:0x3ce218@(irb):2>>
>> a + 3
=> 4
>> a
=> 1
```

`Lazy::Promise` objects are very cool, but a little bit sneaky. Because they essentially work the same as the evaluated object once the block is run once, it's hard to know that you're actually working with a proxy object. But if we dig deep, we can find the truth:

```
>> a.class
=> Fixnum
```

```
>> a.__class__
=> Lazy::Promise
```

If we try out the same examples we did before with `Cell`, you'll see it works as expected:

```
>> cell = Cell.new("Chunky Bacon\nIs Very\nDelicious")
>> cell.width = 1000
=> 1000
>> cell.to_s
=> "Cell(1000x30)"
>> cell.height = 500
=> 500
>> cell.to_s
=> "Cell(1000x500)"
```

Seeing that the output is the same, we can be satisfied knowing that our *lazy.rb* solution will do the trick, while making our code look a little cleaner. However, we shouldn't just pass the work that `promise()` is doing off as magic—it is worth looking at both for enrichment and because it's genuinely cool code.

I've gone ahead and simplified the implementation of `Lazy::Promise` a bit, removing some of the secondary features while still preserving the core functionality. We're going to look at the naive implementation, but please use this only for studying. The official version from MenTaLguY can handle thread synchronization and does much better error handling than what you will see here, so that's what you'll want if you plan to use *lazy.rb* in real code.

That all having been said, here's how you'd go about implementing a basic `Promise` object on Ruby 1.9:

```
module NaiveLazy
  class Promise < BasicObject

    def initialize(&computation)
      @computation = computation
    end

    def __result__
      if @computation
        @result       = @computation.call
        @computation  = nil
      end

      @result
    end

    def inspect
      if @computation
        "#<NaiveLazy::Promise computation=#{ @computation.inspect }>"
      else
        @result.inspect
      end
    end
```

```
    def respond_to?( message )
      message = message.to_sym
      [:__result__, :inspect].include?(message) ||
        __result__.respond_to? message
    end

    def method_missing(*a, &b)
      __result__.__send__(*a, &b)
    end

  end
end
```

Though compact, this might look a little daunting in one big chunk, so let's break it down. First, you'll notice that `NaiveLazy::Promise` inherits from `BasicObject` rather than `Object`:

```
module NaiveLazy
  class Promise < BasicObject

  end
end
```

`BasicObject` omits most of the methods you'll find on `Object`, making it so you don't need to explicitly remove those methods on your own in order to create a proxy object. This in effect gives you a blank slate object, which is exactly what we want for our current purposes.

The proxy itself works through `method_missing`, handing virtually all messages to the result of `Promise#__result__`:

```
module NaiveLazy
  class Promise < BasicObject
    def method_missing(*a, &b)
      __result__.__send__(*a, &b)
    end
  end
end
```

For the uninitiated, this code essentially just allows `promise.some_function` to be interpreted as `promise.__result__.some_function`, which makes sense when we recall how things worked in `Cell`:

```
>> cell.width
=> #<Lazy::Promise computation=#<Proc:...>>
>> cell.width + 10
=> 114
```

`Lazy::Promise` knew how to evaluate the computation and then pass your message to its result. This `method_missing` trick is how it works under the hood. When we go back and look at how `__result__` is implemented, this becomes even more clear:

```
module NaiveLazy
  class Promise < BasicObject
```

```
    def initialize(&computation)
      @computation = computation
    end

    def __result__
      if @computation
        @result       = @computation.call
        @computation  = nil
      end

      @result
    end

    def method_missing(*a, &b)
      __result__.__send__(*a, &b)
    end

  end
end
```

When we create a new `promise`, it stores a code block to be executed later. Then, when you call a method on the `promise` object, `method_missing` runs the `__result__` method. This method checks to see whether there is a `Proc` object in `@computation` that needs to be evaluated. If there is, it stores the return value of that code, and then wipes out the `@computation`. Further calls to `__result__` return this value immediately.

On top of this, we add a couple of methods to make the proxy more well behaved, so that you can fully treat an evaluated promise as if it were just an ordinary value:

```
module NaiveLazy
  class Promise < BasicObject

    # ...

    def inspect
      if @computation
        "#<NaiveLazy::Promise computation=#{ @computation.inspect }>s"
      else
        @result.inspect
      end
    end

    def respond_to?( message )
      message = message.to_sym
      [:__result__, :inspect].include?(message) or
        __result__.respond_to? message
    end

    # ...
  end
end
```

I won't go into much detail about this code—both of these methods essentially just forward everything they can to the evaluated object, and are nothing more than

underplumbing. However, because these round out the full object, you'll see them in action when we take our new `NaiveLazy::Promise` for a spin:

```
>> num = NaiveLazy::Promise.new { 3 }
=> #<NaiveLazy::Promise computation=#<Proc:0x3cfd98@(irb):2>
>> num + 100
=> 103
>> num
=> 3
>> num.respond_to?(:times)
=> true
>> num.class
=> Fixnum
```

So what we have here is an implementation of a proxy object that doesn't produce an exact value until it absolutely has to. This proxy is fully transparent, so even though `num` here is actually a `NaiveLazy::Promise` instance, not a `Fixnum`, other objects in your system won't know or care. This can be pretty handy when delaying your calculations until the last possible moment is important.

The reason I showed how to implement a `promise` is to give you a sense of what the primitive tools in Ruby are capable of doing. We've seen blocks used in a lot of different ways throughout Ruby, but this particular case might be easy to overlook. Another interesting factor here is that although this concept belongs to the functional programming paradigm, it is also easy to implement using Ruby's object-oriented principles.

As we move on to look at other techniques in this chapter, keep this general idea in mind. Much will be lost in translation if you try to directly convert functional concepts into Ruby, but by playing to Ruby's strengths, you can often preserve the idea without things feeling alien.

We're about to look at some other things that are handy to have in your tool belt, but before we do that, I'll reiterate some key points about lazy evaluation in Ruby:

- Lazy evaluation is useful when you have some code that may never need to be run, or would best be run as late as possible, especially if this code is expensive computationally. If you do not have this need, it is better to do without the overhead.
- All code blocks in Ruby are lazy, and are not executed until explicitly called.
- For simple needs, you can build attribute accessors for your objects that avoid running a calculation until they are called, storing the result in an instance variable once it is executed.
- MenTaLguY's *lazy.rb* provides a more comprehensive solution for lazy evaluation, which can be made to be thread-safe and is generally more robust than the naive example shown in this chapter.

We'll now move on to the sticky issue of state maintenance and side effects, as this is a key aspect of functional programming that goes a little against the grain of traditional Ruby code.

Minimizing Mutable State and Reducing Side Effects

Although Ruby is object-oriented, and therefore relies heavily on mutable state, we can write nondestructive code in Ruby. In fact, many of our `Enumerable` methods are inspired by this.

For a trivial example, we can consider the use case for `Enumerable#map`. We could write our own naive map implementation rather easily:

```
def naive_map(array)
  array.each_with_object([]) { |e, arr| arr << yield(e) }
end
```

When we run this code, it has the same results as `Enumerable#map`, as shown here:

```
>> a = [1,2,3,4]
=> [1, 2, 3, 4]
>> naive_map(a) { |x| x + 1 }
=> [2, 3, 4, 5]
>> a
=> [1, 2, 3, 4]
```

As you can see, a new array is produced, rather than modifying the original array. In practice, this is how we tend to write side-effect-free code in Ruby. We traverse our original data source, and then build up the results of a state transformation in a new object. In this way, we don't modify the original object. Because `naive_map()` doesn't make changes to anything outside of the function, we can say that this code is side-effect-free.

However, this code still uses mutable state to build up its return value. To truly make the code stateless, we'd need to build a new array every time we append a value to an array. Notice the difference between these two ways of adding a new element to the end of an array:

```
>> a
=> [1, 2, 3, 4]
>> a = [1,2,3]
=> [1, 2, 3]
>> b = a << 1
=> [1, 2, 3, 1]
>> a
=> [1, 2, 3, 1]

>> c = a + [2]
=> [1, 2, 3, 1, 2]
>> b
=> [1, 2, 3, 1]
>> a
=> [1, 2, 3, 1]
```

It turns out that `Array#<<` modifies its receiver, and `Array#+` does not. With this knowledge, we can rewrite `naive_map` to be truly stateless:

```
def naive_map(array, &block)
  return [] if array.empty?
  [ yield(array[0]) ] + naive_map(array[1..-1], &block)
end
```

This code works in a different way, building up the result set by calling itself repeatedly, resulting in something like this:

```
[1,2,3,4] => [2], [2,3,4] => [2] + [3], [3,4] => [2] + [3] + [4], [4] =>
   [2] + [3] + [4] + [5] => [2,3,4,5]
```

Depending on your taste for recursion, you may find this solution beautiful or scary. In Ruby, recursive solutions may look elegant and have appeal from a purist's perspective, but when it comes to their drawbacks, the pragmatists win out. Other languages optimize for this sort of thing, but Ruby does not, which is made obvious by this benchmark:

```
require "benchmark"

def naive_map(array, &block)
  new_array = []
  array.each { |e| new_array << block.call(e) }
  return new_array
end

def naive_map_recursive(array, &block)
  return [] if array.empty?
  [ yield(array[0]) ] + naive_map_recursive(array[1..-1], &block)
end

N = 100_000

Benchmark.bmbm do |x|
  a = [1,2,3,4,5]

  x.report("naive map") do
    N.times { naive_map(a) { |x| x + 1 } }
  end

  x.report("naive map recursive") do
    N.times { naive_map_recursive(a) { |x| x + 1 } }
  end
end

# Outputs:

sandal:fp $ ruby naive_map_bench.rb
Rehearsal -------------------------------------------------------
naive map             0.370000   0.010000   0.380000 (  0.373221)
naive map recursive   0.530000   0.000000   0.530000 (  0.539722)
---------------------------------------------- total: 0.910000sec

                          user     system      total        real
naive map             0.360000   0.000000   0.360000 (  0.369269)
naive map recursive   0.530000   0.000000   0.530000 (  0.538872)
```

Even though our functions are somewhat trivial, we see our recursive solution performing significantly slower than the iterative one. The reason behind this is the very high cost of method dispatch in Ruby. This means that despite the identical complexity between our iterative and recursive solutions, the latter can quickly become a performance nightmare. If we use a larger dataset, we can see this only exacerbates the problem:

```
N = 100_000

Benchmark.bmbm do |x|
  a = [1,2,3,4,5] * 20

  x.report("naive map") do
    N.times { naive_map(a) { |x| x + 1 } }
  end

  x.report("naive map recursive") do
    N.times { naive_map_recursive(a) { |x| x + 1 }  }
  end
end

# output

sandal:fp $ ruby naive_map_bench.rb
Rehearsal ------------------------------------------------------
naive map             4.360000   0.020000   4.380000 (  4.393069)
naive map recursive   9.420000   0.030000   9.450000 (  9.498580)
--------------------------------------------- total: 13.830000sec

                          user     system      total        real
naive map             4.350000   0.010000   4.360000 (  4.382038)
naive map recursive   9.420000   0.050000   9.470000 (  9.532602)
```

An important thing to remember is that any recursive solution can be rewritten iteratively. We can actually build an iterative, stateless, naive map without much extra effort:

```
def naive_map_via_inject(array, &block)
  array.inject([]) { |s,e| [ yield(e) ] + s }
end
```

`Enumerable#inject` is a favorite feature among Rubyists for accumulation. The way that it works is by passing two objects into the block: the base object and the current element. After each step through the iteration, the return value of the block becomes the new base. Essentially, this code is doing the same thing our recursive code did, without the recursion. We can take a quick look at the benchmarks now, expecting some improvement by cutting out all those expensive recursive method calls:

```
N = 100_000

Benchmark.bmbm do |x|
  a = [1,2,3,4,5] * 20

  x.report("naive map") do
    N.times { naive_map(a) { |x| x + 1 } }
  end
```

```
  x.report("naive map recursive") do
    N.times { naive_map_recursive(a) { |x| x + 1 }  }
  end

  x.report("naive map via inject") do
    N.times { naive_map_via_inject(a) { |x| x + 1 } }
  end
end

# Output

sandal:fp $ ruby naive_map_bench.rb
Rehearsal ------------------------------------------------------
naive map              4.370000   0.030000   4.400000 (  4.458491)
naive map recursive    9.730000   0.090000   9.820000 ( 10.128538)
naive map via inject   7.550000   0.070000   7.620000 (  7.766988)
---------------------------------------------- total: 21.840000sec

                           user     system      total        real
naive map              4.360000   0.020000   4.380000 (  4.413264)
naive map recursive    9.480000   0.050000   9.530000 (  9.553978)
naive map via inject   7.420000   0.050000   7.470000 (  7.509197)
```

Do these numbers surprise you? As we expected, our `inject`-based solution is much faster than our recursive solution, but why is it so much slower than our dumb brute force and ignorance approach?

The reason behind this is one of the key roadblocks that prevent us from writing stateless code in Ruby. In order to solve this problem without modifying any objects, we need to create a new object every single time an element gets added to the array. As you may have guessed, objects are large in Ruby, and constructing them is a slow process. What's more, if we don't store any of these intermediate values, we risk getting the garbage collector churning frequently to kill off our discarded objects.

Avoiding side effects is different than avoiding mutable state entirely. That's the key point to take away from what we just looked at here. In Ruby, as long as it makes sense to do so, avoiding side effects is a good thing. It reduces the possibility for unexpected bugs much in the same way that avoiding the use of global variables does. However, avoiding the use of mutable state definitely depends more on your individual situation.

We showed two examples that avoided the use of mutable state, both of which might look appealing to people who enjoy functional programming style. However, we saw that their performance was abysmal, and because something like `Enumerable#map` tends to be used in a tight loop, this is a bad time to trade performance for aesthetic value.

However, in other situations, the trade-off may be tipped more in the other direction. If the stateless (possibly recursive) code looks better than other solutions, and performance is not a major concern, don't be afraid to write your code in the more elegant way.

In general, remember the following things:

- The simple way to avoid side effects in Ruby when transforming one object to another is to create a new object, and then populate it by iterating over your original object performing the necessary state transformations.
- You can write stateless code in Ruby by creating new objects every time you perform an operation, such as `Array#+`.
- Recursive solutions may aid in writing simple stateless solutions, but incur a major performance penalty in Ruby.
- Creating too many objects can create performance problems as well, so it is important to find the right balance, and to remember that side effects can be avoided without making things fully stateless.

We'll now move on from how you structure individual functions in your code to how you can organize the larger chunks. So let's take a look at what we can learn from modular organization, and how we can mix it in with our object-oriented code.

Modular Code Organization

In many functional languages, it is possible to group together your related functions using a module. However, we typically think of something different when we think of modules in Ruby:

```
class A

  include Enumerable

  def initialize(arr)
    @arr = arr
  end

  def each
    @arr.each { |e| yield(e) }
  end

end

>> A.new([1,2,3]).map { |x| x + 1 }
=> [2, 3, 4]
```

Here, we've included the `Enumerable` module into our class as a mixin. This enables shared implementation of functionality between classes, but is a different concept than modular code organization in general.

What we really want is a collection of functions unified under a single namespace. As it turns out, Ruby has that sort of thing, too! Although the `Math` module can be mixed into classes similar to the way we've used `Enumerable` here, you can also use it on its own:

```
>> Math.sin(Math::PI / 2)
=> 1.0
>> Math.sqrt(4)
=> 2.0
```

So, how'd they do that? One way is to use `module_function`:

```
module A
  module_function

  def foo
    "This is foo"
  end

  def bar
    "This is bar"
  end
end
```

We can now call these functions directly on the module, as you can see here:

```
>> A.foo
=> "This is foo"
>> A.bar
=> "This is bar"
```

You won't need anything more for most cases in which you want to execute functions on a module. However, this approach does come with some limitations, because it does not allow you to use private functions:

```
module A
  module_function

  def foo
    "This is foo calling baz: #{baz}"
  end

  def bar
    "This is bar"
  end

  private

  def baz
    "hi there"
  end
end
```

Though it seems like our code is fairly intuitive, we'll quickly run into an error once we try to call `A.foo`:

```
>> A.foo
NameError: undefined local variable or method 'baz' for A:Module
  from (irb):33:in 'foo'
  from (irb):46
  from /Users/sandal/lib/ruby19_1/bin/irb:12:in '<main>'
```

For some cases, not being able to access private methods might not be a big deal, but for others, this could be a major issue. Luckily, if we think laterally, there is an easy workaround.

Modules in Ruby, although they cannot be instantiated, are in essence ordinary objects. Because of this, there is nothing stopping us from mixing a module into itself:

```
module A
  extend self

  def foo
    "This is foo calling baz: #{baz}"
  end

  def bar
    "This is bar"
  end

  private

  def baz
    "hi there"
  end
end
```

Once we do this, we get the same effect as `module_function` without the limitations:

```
>> A.foo
=> "This is foo calling baz: hi there"
```

We aren't sacrificing encapsulation here, either. We will still get an error if we try to call `A.baz` directly:

```
>> A.baz
NoMethodError: private method 'baz' called for A:Module
  from (irb):65
  from /Users/sandal/lib/ruby19_1/bin/irb:12:in '<main>'
```

Using this trick of extending a module with itself provides us with a structure that isn't too different (at least on the surface) from the sort of modules you might find in functional programming languages. But aside from odd cases such as the `Math` module, you might wonder when this technique would be useful.

For the most part, classes work fine for encapsulating code in Ruby. Traditional inheritance combined with the powerful mixin functionality of modules covers most of the bases just fine. However, there are definitely cases in which a concept isn't big enough for a class, but isn't small enough to fit in a single function.

I ran into this issue recently in a Rails app I was working on. I was implementing user authentication and needed to first test the database via ActiveRecord, and fall back to LDAP when a user didn't have an account set up in the application.

Without getting into too much detail, the basic structure for my authentication routine looked like this:

```
class User < ActiveRecord::Base

  # other model code omitted

  def self.authenticate(login, password)
    if u = find_by_login(login)
      u.authenticated?(password) ? u : nil
    else
      ldap_authenticate(login, password)
    end
  end

end
```

LDAP authentication was to be implemented in a private class method, which seemed like a good idea at first. However, as I continued to work on this, I found myself writing a very huge function that represented more than a page of code. As I knew there would be no way to keep this whole thing in my head easily, I proceeded to break things into more helper methods to make things clearer. Unfortunately, this approach didn't work as well as I had hoped.

By the end of this refactoring, I had racked up all sorts of strange routines on `User`, with names such as `initialize_ldap_conn`, `retrieve_ldap_user`, and so on. A well-factored object should do one thing and do it well, and my `User` model seemed to know much more about LDAP than it should have to. The solution was to break this code off into a module, which was only a tiny change to the `User.authenticate` method:

```
def self.authenticate(login, password)
  if u = find_by_login(login) # need to get the salt
    u.authenticated?(password) ? u : nil
  else
    LDAP.authenticate(login, password)
  end
end
```

By substituting the private method call on the `User` model with a modular function call on `User::LDAP`, I was able to define my function and its private helpers in a place that made more sense. The module ended up looking something like this:

```
module LDAP

  extend self

  def authenticate(username, password)
    connection = initialize_ldap_connection
    retrieve_ldap_user(username, password, connection)
  rescue Net::LDAP::LdapError => e
    ActiveRecord::Base.logger.debug "!!! LDAP Error: #{e.message} !!!"
    false
  end

  private
```

```
  def initialize_ldap_connection
    #...
  end

  def retrieve_ldap_user(username, password, connection)
    #...
  end

end
```

This definitely cleaned up the code and made it easier to follow, but it had additional benefits as well. It introduced a clear separation of concerns that helped make testing much easier. It also left room for future expansion and modification without tight coupling.

Of course, if we end up needing to do more than simply authenticate a user against the LDAP database, this module will need to go. As soon as you see the same argument being passed to a bunch of functions, you might be running into a situation where some persistence of state wouldn't hurt. You can probably see how we'd spend a lot of time passing around usernames and connection objects if this code grew substantially bigger.

The good news is, if need arises for expansion down the line, converting code that has been organized into a module is somewhat trivial. Here's how we'd do it with the LDAP module:

```
class LDAP

  def self.authenticate(username, password)
    user = new(username, password)
    user.authenticate(password)
  rescue Net::LDAP::LdapError => e
    ActiveRecord::Base.logger.debug "!!! LDAP Error: #{e.message} !!!"
    false
  end

  def initialize(username)
    @connection = initialize_ldap_connection
    @username   = username
  end

  def authenticate(password)
    #...
  end

  private

  def initialize_ldap_connection
    #...
  end

end
```

As you can see, the difference is somewhat minimal, and no changes need to be made to your user model. Object-oriented purists may even prefer this approach all around, but there is a certain appeal to the minimalism of the modular approach shown earlier.

Even though this latest iteration moves to a more object-oriented approach, there is still modular appeal to it. The ease of creating class methods in Ruby contributes highly to that, as it makes it possible for this to look like modular code even if it's building a new instance of the LDAP module each time.

Although it is not advisable to try extra hard to use this technique of modular code design in every possible situation, it is a neat organizational approach that is worth knowing about. Here are a few things to watch for that indicate this technique may be the right way to go:

- You are solving a single, atomic task that involves lots of steps that would be better broken out into helper functions.
- You are wrapping some functions that don't rely on much common state between them, but are related to a common topic.
- The code is very general and can be used standalone *or* the code is very specific but doesn't relate directly to the object that it is meant to be used by.
- The problem you are solving is small enough where object orientation does more to get in the way than it does to help you.

Because modular code organization reduces the amount of objects you are creating, it can potentially give you a decent performance boost. This offers an incentive to use this approach when it is appropriate.

Memoization

A typical "Hello World" program in functional programming languages is a recursive function that computes the Fibonacci sequence. In Ruby, the trivial implementation looks like this:

```
def fib(n)
  return n if (0..1).include? n
  fib(n-1) + fib(n-2)
end
```

However, you'll feel the pain that is relying on deep recursion in Ruby if you compute even modest values of `n`. On my machine, `fib(30)` computes within a few seconds, but I'm too impatient to even give you a time for `fib(40)`. However, there is a special characteristic of functions like this that makes it possible to speed them up drastically.

In mathematics, a function is said to be well defined if it consistently maps its input to exactly one output. This is obviously true for `fib(n)`, as `fib(6)` will always return `8`, no matter how many times you compute it. This sort of function is distinct from one that is not well defined, such as the following:

```
def mystery(n)
  n + rand(1000)
end
```

If we run this code a few times with the same `n`, we see there isn't a unique relationship between its input and output:

```
>> mystery(6)
=> 928
>> mystery(6)
=> 671
>> mystery(6)
=> 843
```

When we have a function like this, there isn't much we can assume about it. However, well-defined functions such as `fib(n)` can get a massive performance boost almost for free. Can you guess how?

If your mind wandered to tail-call optimization or rewriting the function iteratively, you're thinking too hard. However, the idea of reducing the amount of recursive calls is on track. As it stands, this code is a bad dream, as `fib(n)` is called five times when `n`=3 and nine times when `n`=4, with this trend continuing upward as `n` gets larger.

The key realization is what I mentioned before: `fib(6)` is always going to be `8`, and `fib(10)` is always going to be `55`. Because of this, we can store these values rather than calculate them repeatedly. Let's give that a shot and see what happens:

```
def fib(n)
  @series[n] ||= fib(n-1) + fib(n-2)
end
```

Huzzah! By simply storing our precalculated values in an array, we can now calculate much deeper into the sequence:

```
>> fib(1000)
=> 43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875
```

What we have done is used a technique called *memoization* to cache the return values of our function based on its input. Because we were caching a sequence, it's reasonable to use an array here, but in other cases in which the data is more sparse, a hash may be more appropriate. Let's take a look at some real code where that is the case, to help illustrate the point.

In my PDF generation library Prawn, I provide helper methods that convert from HTML colors to an array of RGB values and back again. Though they're nothing particularly exciting, this is what they look like in action:

```
>> rgb2hex([100,25,254])
=> "6419fe"

>> hex2rgb("6419fe")
=> [100, 25, 254]
```

The implementations of these functions are somewhat simple, and not nearly as computationally expensive as our recursive Fibonacci implementation:

```
def rgb2hex(rgb)
  rgb.map { |e| "%02x" % e }.join
end

def hex2rgb(hex)
  r,g,b = hex[0..1], hex[2..3], hex[4..5]
  [r,g,b].map { |e| e.to_i(16) }
end
```

Although these methods aren't especially complicated, they represent a decent use case for caching via memoization. Colors are likely to be reused frequently and, after they have been translated once, will never change. Therefore, `rgb2hex()` and `hex2rgb()` are well-defined functions.

As it turns out, Ruby's `Hash` is a truly excellent cache object. Before we get into the specifics, take a look at the memoized versions and see if you can figure out for yourself what's going on:

```
def rgb2hex_manual_cache(rgb)
  @rgb2hex ||= Hash.new do |colors, value|
    colors[value] = value.map { |e| "%02x" % e }.join
  end

  @rgb2hex[rgb]
end

def hex2rgb_manual_cache(hex)
  @hex2rgb ||= Hash.new do |colors, value|
    r,g,b = value[0..1], value[2..3], value[4..5]
    colors[value] = [r,g,b].map { |e| e.to_i(16) }
  end

  @hex2rgb[hex]
end
```

Does this example make much sense? If you look at it closely, you can see that the core implementation is still the same—we've just added some extra code to do the caching for us. To do this, we use the block form of `Hash.new`.

You may have used `Hash.new` to define a default value when an unknown key is used. This trick is a relatively simple way to do all sorts of things, including constructing a hash of arrays, or creating a hash for counting things:

```
>> a = Hash.new { |h,k| h[k] = [] }
=> {}
>> a[:foo] << 1
=> [1]
>> a[:foo] << 2
=> [1, 2]
>> a[:bar]
=> []
```

```
>> b = Hash.new { |h,k| h[k] = 0 }
=> {}
>> [:foo, :bar, :foo, :bar, :bar, :bar, :foo].each { |e| b[e] += 1 }
=> [:foo, :bar, :foo, :bar, :bar, :bar, :foo]
>> b
=> {:foo=>3, :bar=>4}
```

However, if we can compute our value based on the key to our hash, we can do more than simply provide default values. For example, it would be easy to create a hash that would multiply any key passed into it by 2:

```
>> doubler = Hash.new { |h,k| h[k] = k * 2 }
=> {}
>> doubler[2]
=> 4
>> doubler[5]
=> 10
>> doubler[10]
=> 20
```

With this in mind, it should be easier to understand the caching in our color conversion code upon a second glance:

```
def rgb2hex_manual_cache(rgb)
  @rgb2hex ||= Hash.new do |colors, value|
    colors[value] = value.map { |e| "%02x" % e }.join
  end

  @rgb2hex[rgb]
end

def hex2rgb_manual_cache(hex)
  @hex2rgb ||= Hash.new do |colors, value|
    r,g,b = value[0..1], value[2..3], value[4..5]
    colors[value] = [r,g,b].map { |e| e.to_i(16) }
  end

  @hex2rgb[hex]
end
```

Here we can see that the input to the function is being used to build up our `Hash`. We initialize the `Hash` once, and then we simply index into it to populate the values. The block is run exactly once per key, and then the cached values are returned thereafter. As a result, we have greatly sped up our function. Let's take a look at some benchmarks to see how much of a boost we get by writing things this way:

```
require "benchmark"

N = 500_000

Benchmark.bmbm do |x|
```

```
  x.report("rgb2hex_uncached") do
    N.times { rgb2hex([100,25,50]) }
  end
  x.report("rgb2hex_manual_cache") do
    N.times { rgb2hex_manual_cache([100,25,50]) }
  end

  x.report("hex2rgb_uncached") do
    N.times { hex2rgb("beaded") }
  end
  x.report("hex2rgb_manual_cache") do
    N.times { hex2rgb_manual_cache("beaded") }
  end

end

sandal:fp $ ruby rgb2hex.rb
Rehearsal ------------------------------------------------------
rgb2hex_uncached       3.560000   0.030000   3.590000 (  3.656217)
rgb2hex_manual_cache   1.030000   0.000000   1.030000 (  1.063319)
hex2rgb_uncached       1.220000   0.010000   1.230000 (  1.240591)
hex2rgb_manual_cache   0.280000   0.000000   0.280000 (  0.303417)
--------------------------------------------- total: 6.130000sec

                           user     system      total        real
rgb2hex_uncached       3.570000   0.040000   3.610000 (  3.733938)
rgb2hex_manual_cache   1.040000   0.010000   1.050000 (  1.055863)
hex2rgb_uncached       1.210000   0.010000   1.220000 (  1.248148)
hex2rgb_manual_cache   0.280000   0.000000   0.280000 (  0.284613)
```

As you can see, the results are pretty convincing. The cached version of the code is several times faster than the uncached one. This means that when running under a tight loop, the memoization can really make a big difference in these functions, and may be worth the minimal noise introduced by adding a `Hash` into the mix.

You might have noticed that the process for doing memoization isn't really case-specific. In nearly every situation, you're going to want to create a simple hash that maps from the input to the output, and you'll want to return that value rather than recalculate it if it exists. Ruby wouldn't be Ruby if we couldn't hack our way around repetition like this, and luckily James Gray is one of the folks who has done exactly that.

James wrote a nice little module called `Memoizable`, which is designed to abstract the task of creating a cache to the point at which you simply mark each function that should be memoized similar to the way you mark something public or private.

Let's take a look at this in action, before digging deeper:

```
include Memoizable

def rgb2hex(rgb)
  rgb.map { |e| "%02x" % e }.join
end
```

```
memoize :rgb2hex

def hex2rgb(hex)
  r,g,b = hex[0..1], hex[2..3], hex[4..5]
  [r,g,b].map { |e| e.to_i(16) }
end

memoize :hex2rgb
```

That's really all there is to it. `Memoizable` works by making a copy of your function, renaming it as `__unmemoized_method_name__`, and then injects its automatic caching in place of the original function. That means that when we call `rgb2hex()` or `hex2rgb()`, we'll now be hitting the cached versions of the functions.

This is pretty exciting, as it means that for well-defined functions, you can use `Memoizable` to get a performance boost without even modifying your underlying implementation. Let's take a look at how this approach stacks up performance-wise when compared to the manual caching from before:

```
require "benchmark"

N = 500_000

Benchmark.bmbm do |x|

  x.report("rgb2hex (Memoizable)") do
    N.times { rgb2hex([100,25,50]) }
  end
  x.report("rgb2hex_manual_cache") do
    N.times { rgb2hex_manual_cache([100,25,50]) }
  end
  x.report("rgb2hex_uncached") do
    N.times { __unmemoized_rgb2hex__([100,25,50]) }
  end

  x.report("hex2rgb (Memoizable)") do
    N.times { hex2rgb("beaded") }
  end
  x.report("hex2rgb_manual_cache") do
    N.times { hex2rgb_manual_cache("beaded") }
  end
  x.report("hex2rgb_uncached") do
    N.times { __unmemoized_hex2rgb__("beaded") }
  end

end

sandal:fp $ ruby rgb2hex.rb
Rehearsal -----------------------------------------------------
rgb2hex (Memoizable)   1.750000   0.010000   1.760000 (  1.801235)
rgb2hex_manual_cache   1.040000   0.010000   1.050000 (  1.067790)
rgb2hex_uncached       3.580000   0.020000   3.600000 (  3.680780)
hex2rgb (Memoizable)   0.990000   0.010000   1.000000 (  1.021821)
hex2rgb_manual_cache   0.280000   0.000000   0.280000 (  0.287521)
```

```
hex2rgb_uncached       1.210000   0.010000   1.220000 (  1.247875)
------------------------------------------------ total: 8.910000sec

                           user     system      total        real
rgb2hex (Memoizable)   1.760000   0.010000   1.770000 (  1.803120)
rgb2hex_manual_cache   1.040000   0.000000   1.040000 (  1.066625)
rgb2hex_uncached       3.600000   0.030000   3.630000 (  3.871221)
hex2rgb (Memoizable)   0.990000   0.010000   1.000000 (  1.017367)
hex2rgb_manual_cache   0.280000   0.000000   0.280000 (  0.283920)
hex2rgb_uncached       1.220000   0.010000   1.230000 (  1.248152)
```

Although `Memoizable` is predictably slower than our raw implementation, it is still cooking with gas when compared to the uncached versions of our functions. What we are seeing here is the overhead of an additional method call per request, so as the operation becomes more expensive, the cost of `Memoizable` actually gets lower. Also, if we look at things in terms of work versus payout, `Memoizable` is the clear winner, due to its ability to transparently hook itself into your functions.

I could stop here and move on to the next topic, but similar to when we looked into the belly of *lazy.rb* earlier in this chapter, I can't resist walking through and explaining some cool code. I am hoping that as you read through this book, you will find that things that seem magical on the surface have a clear and easy-to-understand implementation under the hood, and `Memoizable` is a perfect example for that.

Here is the full implementation of the module:

```
module Memoizable
  def memoize( name, cache = Hash.new )
    original = "__unmemoized_#{name}__"

    ([Class, Module].include?(self.class) ? self : self.class).class_eval do
      alias_method original, name
      private      original
      define_method(name) { |*args| cache[args] ||= send(original, *args) }
    end
  end
end
```

We see that the `memoize()` method takes a method name and an optional cache object, which defaults to an empty `Hash` if none is provided. The code then does some logic to determine whether you are using the top-level object (as we were), or just working in the context of an ordinary object. Once it figures that out, what remains is a simple `class_eval` block, which does what we talked about before. The original method is renamed, and then a new method is created that caches the return values of the original function based on the input arguments. Despite how powerful this code is, we see that it is of only marginally higher complexity than our handwritten caching.

Of course, this doesn't have a whole lot to do with functional programming techniques, except for showing you how you can abstract them into reusable constructs. In case this distracted you a bit, here are the things to remember about memoization before moving on to the next section:

- Functions that are well defined, where a single input consistently produces the same output, can be cached through memoization.
- Memoization often trades CPU time for memory, storing results rather than recalculating them. As a result, memoization is best used when memory is cheap and CPU time is costly, and not the other way around. In some cases, even when the memory consumption is negligible, the gains can be substantial. We can see this in the `fib(n)` example, which is transformed from an exponential algorithm to a linear one simply by storing the intermediate calculations.
- When coding your own solution, `Hash.new`'s block form can be a very handy way of putting together a simple caching object.
- James Gray's `Memoizable` module makes it trivial to introduce memoization to well-defined functions without directly modifying their implementations, but incurs a small cost of indirection over an explicit caching strategy.

When dealing with sequences that can be generated from previous values, memoization isn't the only game in town. We're now going to take a look at how to build upon the concept of lazy evaluation to form very interesting structures known as infinite lists.

Infinite Lists

Infinite lists (also known as lazy streams) provide a way to represent arbitrary sequences that can be traversed by applying a certain function that gets to you the next element for any given element in the list. For example, if we start with any even number, we can get to the next one in the sequence by simply adding 2 to our original element.

Before we look at more complicated examples, let's take a look at how we can represent the even sequence of numbers this way, using a simple Ruby object:

```
module EvenSeries
  class Node
    def initialize(number=0)
      @value = number
      @next  = lambda { Node.new(number + 2) }
    end

    attr_reader :value

    def next
      @next.call
    end
  end
end

e = EvenSeries::Node.new(30)
10.times do
  p e.value
  e = e.next
end
```

When we run this code, we get the following output:

```
30
32
34
36
38
40
42
44
46
48
```

The implementation should be mostly self-explanatory. An `EvenSeries::Node` is nothing more than a number and a `Proc` that is designed to add 2 to that number and construct a new `Node`. What we end up with is something that looks similar to a linked list, and we can clearly see what happens when we take 10 steps forward from 30 through the even numbers.

The key innovation is that we've turned an external iteration and state transformation into an internal one. The benefits of this technique will become more clear in later examples, but for now, it's worth noting that this is the key motivation for creating such a construct.

Although this example shows you the simplest thing that could possibly work when it comes to infinite lists, we can benefit from using a more generalized, more feature-complete structure for this task. Rather than rolling our own, I'm going to walk you through some of the examples for James Gray's `LazyStream` implementation, many of which you may have already seen in his Higher-Order Ruby blog series (*http://blog.grayproductions.net/categories/higherorder_ruby*). His focus was on showing how to build an infinite list structure; mine will be on how to make use of them.

The following is a trivial example of using *lazy_stream.rb*, doing simple iteration over a range of numbers:

```
require "lazy_stream"

def upto( from, to )
  return if from > to
  lazy_stream(from) { upto(from + 1, to) }
end
upto(3, 6).show  # => 3 4 5 6

def upfrom( start )
  lazy_stream(start) { upfrom(start + 1) }
end
upfrom(7).show(10)  # => 7 8 9 10 11 12 13 14 15 16
```

As you can see here, `lazy_stream()` is just creating recursive calls that build up new elements that in turn know how to get to their next element. What is neat is that as the name suggests, these things are evaluated lazily. When we call `upto(3, 6)`, a `Proc` is set up to carry out this task for us, but it's not actually executed until we tell it to show us

the results. Something similar is happening when we call `upfrom(7)`. Though this example is fairly basic, it gives us a jumping point into `lazy_stream`. We can make things more interesting by introducing classes into the mix:

```
require "lazy_stream"

class Upto < LazyStream::Node
  def initialize( from, to )
    if from > to
      super(nil, &nil)
    else
      super(from) { self.class.new(from + 1, to) }
    end
  end
end
Upto.new(3, 6).show  # => 3 4 5 6

class Upfrom < LazyStream::Node
  def initialize( from )
    super(from) { self.class.new(from + 1) }
  end
end
Upfrom.new(7).show(10)  # => 7 8 9 10 11 12 13 14 15 16
```

Though this code looks a bit more complex, it is also more flexible. What we have done is created our own custom `LazyStream::Node` objects, which, as you can see, accomplish the same thing as we did before but without relying on a generic constructor.

`LazyStream::Node` objects are enumerable, and this lets us do all sorts of fun stuff. The following code illustrates this by constructing a simple step iterator:

```
require "lazy_stream"

class Step < LazyStream::Node
  def initialize( step, start = 1 )
    super(start) { self.class.new(step, start + 1) }

    @step = step
  end

  def next_group( count = 10 )
    limit!(count).map { |i| i * @step }
  end
end

evens = Step.new(2)

puts "The first ten even numbers are:"
puts evens.next_group.join(" ")  # => 2 4 6 8 10 12 14 16 18 20

# later...

puts
puts "The next ten even numbers are:"
puts evens.next_group.join(" ")  # => 22 24 26 28 30 32 34 36 38 40
```

```
puts
puts "The current index for future calculations is:"
puts evens.current # => 21
```

Here is where the benefit of an internal iterator becomes clear. When we work with a `Step` object once it's been set up, we don't really care what its underlying function is from element to element. However, we can still work with it and pull out groups of elements as needed, keeping track of where we are in the list as we go along.

If we wanted to jump up in steps of three instead of two, you can see that the code changes are minimal:

```
threes = Step.new(3)

puts "The first ten multiples of 3 are"
puts threes.next_group.join(" ")  # => 3 6 9 12 15 18 21 24 27 30

# later...

puts
puts "The next ten are:"
puts threes.next_group.join(" ")  # => 33 36 39 42 45 48 51 54 57 60

puts
puts "The current index for future calculations is:"
puts threes.current # => 21
```

Though this stuff is pretty cool, things get a whole lot more interesting when you mix filters and transforms into the mix:

```
require "lazy_stream"

def letters( letter )
  lazy_stream(letter) { letters(letter.succ) }
end

letters("a").filter(/[aeiou]/).show(10)          # => a e i o u aa ab ac ad ae
letters("a").filter { |l| l.size == 2 }.show(3)  # => aa ab ac

letters("a").transform { |l| l + "..." }.show(3)  # => a... b... c...
```

Here, we're walking over successive ASCII string values. In the very first example, we ask `lazy_stream` to nab us the first 10 values that contain a vowel. In the second example, we ask for the first three strings of length two. In the third example, we ask to show the first three values with `...` appended to them. In all three cases, `lazy_stream` is happy to provide us with what we'd expect.

To really get a sense for how nice this is, go ahead and write out a solution to these three problems using normal iterators and conditionals. Then compare your solution to using `lazy_stream`. What you will most likely find is that because `lazy_stream` internalizes a lot of the logic for us, it is more pleasant to work with and generally more expressive.

Before we move on, I'd like to show one more trick that `lazy_stream` has up its sleeve, and that is the ability to build up an infinite list recursively:

```
require "lazy_stream"

class Powers < LazyStream::Node
  def initialize( of, start = 1 )
    super(start) { self.class.new(of, start * of) }
  end
end

powers_of_two = Powers.new(2)
powers_of_two.show(10)  # => 1 2 4 8 16 32 64 128 256 512
```

We have caught glimpses of this technique in other examples, but this one shows clearly how you can build up a list based on its previous values. Here we construct the powers of two by repeatedly constructing new `LazyStream::Node` objects with values twice as much as their predecessors.

This turns out to be a simple, expressive way to traverse, filter, and transform lists that are linked together by a function that binds one element to the next. However, if this stuff is making your head hurt, you need to remember just the following things:

- Infinite lists essentially consist of nodes that contain a value along with a procedure that will transform that value into the next element in the sequence.
- Infinite lists are lazily evaluated, and thus are sometimes called lazy streams.
- An infinite list might be an appropriate structure to use when you need to iterate over a sequential list in groups at various points in time, or if you have a general function that can be tweaked by some parameters to fit your needs.
- For data that is sparse, memoization might be a better technique than using an infinite list.
- When you need to do filtering or state transformation on a long sequence of elements that have a clear relationship from one to the next, a lazy stream might be the best way to go.
- JEG2's *lazy_stream.rb* provides a generalized implementation of infinite lists that is worth taking a look at if you have a need for this sort of thing.

If you're still hanging on this far into the chapter, you're either really enjoying this weird functional stuff, or you're a very determined reader. Either way, you will be happy to know that all of the hard parts are over. Before we close the chapter and make some conclusions, let's take a quick look at one of the most simple but fundamentally useful things Ruby has borrowed from its functional peers: higher-order procedures.

Higher-Order Procedures

Throughout this whole chapter, we've been taking something for granted that not all languages can: in Ruby, a `Proc` is just another object. This means we can sling these

chunks of code around as if they were any other value, which results in a whole lot of powerful functionality.

This feature is so ingrained in Ruby development, and so well integrated into the system, that it's easy to overlook. However, I could not in good conscience wrap up a chapter on functional programming techniques without at least showing a simple example of higher-order procedure.

A function is said to be a higher-order function if it accepts another function as input or returns a function as its output. We see a lot of the former in Ruby; basically, we'd see this any time we provide a code block to a method. But functions that return functions might be a bit less familiar to those who are new to the language.

Through closures, we can use a function to essentially build up customized procedures on the fly. I could show some abstract examples or academically exciting functionality such as `Proc#curry`, but instead, I decided that I wanted to show you something I use fairly frequently.

If you used Rails before Ruby 1.9, you probably were familiar with the "innovation" of `Symbol#to_proc`. What some smart folks realized is that Ruby's `&block` mechanism actually calls a hook on the underlying object, which could be self-defined. Though `Symbol#to_proc` exists in Ruby 1.9 by default, let's look at what a simple implementation of it would look like in Ruby:

```
class Symbol
  def to_proc
    lambda { |x| x.send(self) }
  end
end
```

As you probably know, this feature allows for some nice syntactic sugar:

```
>> %w[foo bar baz].map(&:capitalize)
=> ["Foo", "Bar", "Baz"]
```

The way that it works should be easy to see from the implementation shown here, but in essence, what happens is that `&:capitalize` invokes `Symbol#to_proc` and then constructs a block like this:

```
lambda { |x| x.send(:capitalize) }
```

This in turn is then treated as a code block, making it functionally identical to the following code:

```
>> %w[foo bar baz].map { |x| x.capitalize }
```

Of course, you've probably seen a hundred blog posts about this feature, so this is not what I'm really trying to show you. What I'd like you to know, and eventually take advantage of, is that `Object#to_proc` is a generic hook. This means `Symbol#to_proc` isn't special, and we can build our own custom objects that do even cooler tricks than it does.

The place I use this functionality all the time is in Rails applications where I need to build up filter mechanisms that do some of the work in SQL, and the rest in Ruby.

Though my filter objects typically grow to be long, complicated, and nasty, their backbone always lies upon something somewhat simple. Here is the general pattern I usually start with:

```
class Filter
  def initialize
    @constraints = []
  end

  def constraint(&block)
    @constraints << block
  end

  def to_proc
    lambda { |e| @constraints.all? { |fn| fn.call(e) } }
  end
end
```

We can then construct a `Filter` object and assign constraints to it on the fly:

```
filter = Filter.new
filter.constraint { |x| x > 10 }
filter.constraint { |x| x.even? }
filter.constraint { |x| x % 3 == 0 }
```

Now, when dealing with an `Enumerable` object, it is easy to filter the data based on our constraints:

```
p (8..24).select(&filter) #=> [12,18,24]
```

As we add more constraints, new blocks are generated for us, so things work as expected:

```
filter.constraint { |x| x % 4 == 0 }

p (8..24).select(&filter) #=> [12,24]
```

As you can see, `Symbol#to_proc` isn't the only game in town. Any object that can meaningfully be reduced to a function can implement a useful `to_proc` method.

Of course, higher-order procedures are not limited to `to_proc` hacks. There are lots of other good uses for functions that return functions, though it is often a relatively rare use case, so it's hard to give you a list of must-have techniques when it comes to this topic.

Because this example was short and sweet, we don't need to go over various guidelines as we have in previous sections. Just keep this trick in the back of your mind, and use it to improve the readability and flexibility of your code.

We've covered a lot of ground here, so it's about time to wrap up this chapter.

Conclusions

Although we covered a number of "functional programming techniques" throughout this chapter, it's fairly plain to see that Ruby isn't a functional programming language. Even if we can force Ruby into being pretty much whatever we want it to be, it's generally not an advisable thing to do.

Nevertheless, we've covered some interesting and useful tricks here that were inspired by techniques used in other languages. By letting Ruby be Ruby and not striving too hard for a direct translation, we end up with code that is actually practical and suitable for use in everyday applications.

My general feeling about applying functional programming techniques in Ruby is mainly that they can be a true source of elegance and simplicity when appropriate. However, judging when it's the right time to bust out some functional goodness rather than going with the more vanilla approach can be difficult, even among seasoned developers. It is entirely possible to make your code too clever, and this often has real penalties in performance or in the ability of other developers to easily learn your code.

Many functional programming techniques result in code that is highly readable at the expense of learnability, at least in the context of Ruby. If you keep an eye on that balance and make it fit within your working environment, you'll be just fine.

Even if some of the techniques seen in this chapter cannot be readily used in a direct way in many cases, there is a lot to be said for thinking differently and looking at your code from another angle. Hopefully, this chapter has helped you do that.