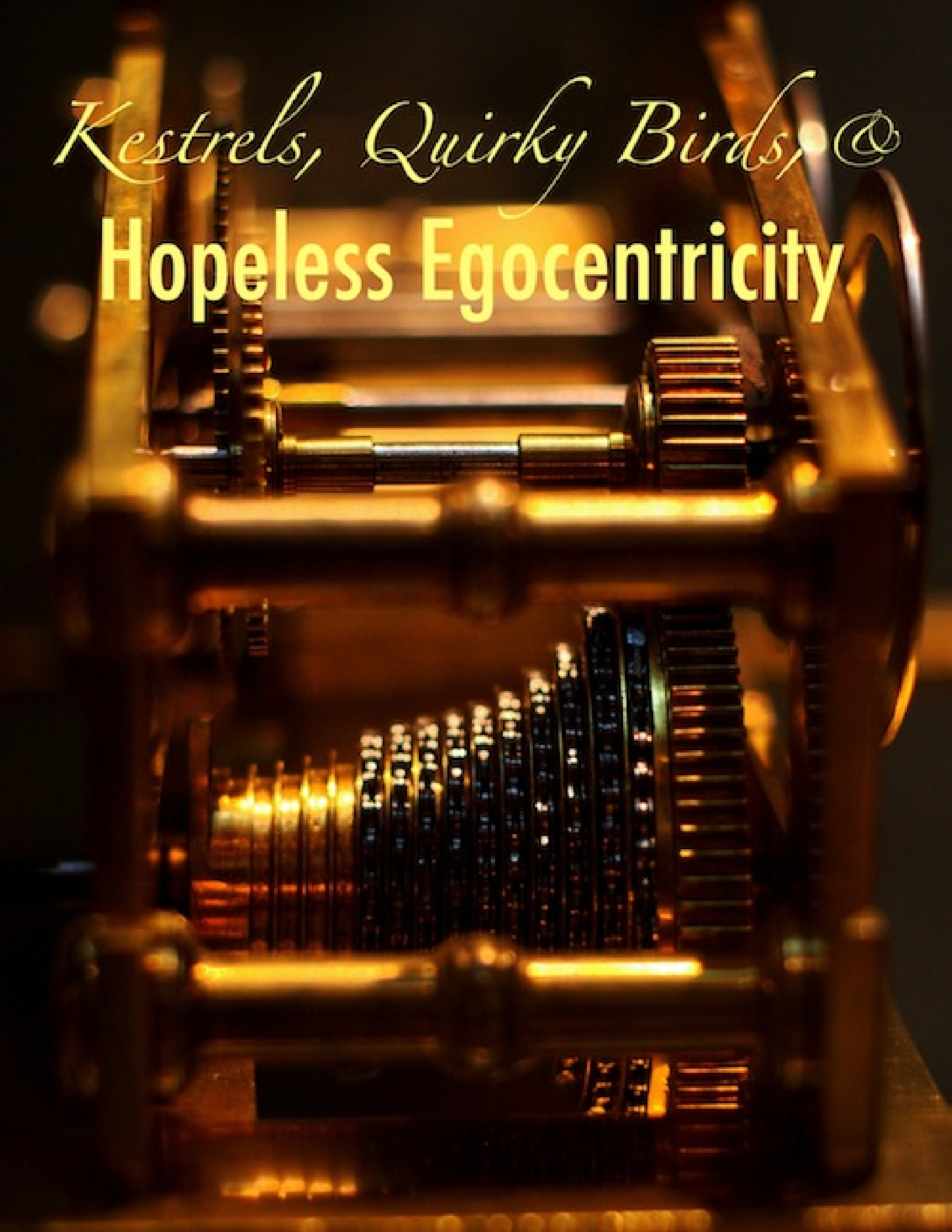
Kestrels, Quirky Birds, &
Hopeless Egocentricity

Kestrels, Quirky Birds, and Hopeless Egocentricity

Raganwald's collected adventures in Combinatory Logic and Ruby Meta-Programming

Reginald Braithwaite

This book is for sale at http://leanpub.com/combinators

This version was published on 2013-10-01

Tweet This Book!

Please help Reginald Braithwaite by spreading the word about this book on Twitter!

The suggested hashtag for this book is #combinators.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#combinators

Also By Reginald Braithwaite

What I've Learned From Failure

How to Do What You Love & Earn What You're Worth as a Programmer

CoffeeScript Ristretto

JavaScript Allongé

Contents

0.1 The MIT License

0.2 Preface

The chapters of this book originally appeared as blog posts. You can still read them online, for free, at http://github.com/raganwald/homoiconic[1]. The original posts were released under the MIT license, you you can pass them around or incorporate them into your own works as you see fit. I decided to publish these essays as an e-book as well as online. This format doesn't replace the original online essays, it's a way to present these essays in a more coherent whole that's easier to read consecutively. I hope you like it.

–Reginald "Raganwald" Braithwaite[2], Toronto, November 2011

[1] http://github.com/raganwald/homoiconic
[2] http://braythwayt.com

1 Introduction

Like the Lambda Calculus, Combinatory Logic[1] is a mathematical notation that is powerful enough to handle set theory and issues in computability.

> Combinatory logic is a notation introduced by Moses Sch□□nfinkel[2] and Haskell Curry[3] to eliminate the need for variables in mathematical logic. It has more recently been used in computer science as a theoretical model of computation and also as a basis for the design of functional programming languages. It is based on combinators. A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments.

In this book, we're going to meet some of the standard combinators, and for each one we'll explore some of its ramifications when writing programs using the Ruby programming language. In Combinatory logic, combinators combine and alter each other, and our Ruby examples will focus on combining and altering Ruby code. From simple examples like the K Combinator and Ruby's `.tap` method, we'll work our way up to meta-programming with aspects and recursive combinators.

about the bird names

When Combinatory Logic was first invented by Haskell Curry, the standard combinators were given upper-case letters. For example, the two combinators needed to express everything in the Lambda Calculus and in Set Theory are the `S` and `K` combinators. In 1985, Raymond Smullyan published To Mock a Mockingbird[4], an exploration of combinatory logic for the recreational layman. Smullyan used a forest full of songbirds as a metaphor, with each of the combinators given the name of a songbird rather than a single letter. For example, the `S` and `K` combinators became the Starling and Kestrel, the `I` combinator became the Idiot bird, and so forth.

These ornithological nicknames have become part of the standard lexicon for combinatory logic.

thanks

There are too many people to name,but amongst the crowd, Alan Smith stands out.

[1] http://en.wikipedia.org/wiki/Combinatory_logic

[2] http://en.wikipedia.org/wiki/Moses_Sch□□nfinkel

[3] http://en.wikipedia.org/wiki/Haskell_Curry

[4] http://www.amazon.com/gp/product/B00A1P096Y/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=B00A1P096Y&linkCode=as2&tag=raganwald001-20

1.1 About this sample

This sample edition of the book includes the first full chapter, “Kestrels.” The full book adds chapters about Thrushes and permuting the order of method pipelining, Cardinals and constructing our own monad-like `maybe` function, Quirky Birds and meta-programming with methods, Bluebirds and Aspect-Oriented Programming, Recursive Combinators, Hopelessly Egocentric birds and the semantics of nil, and more.

2 Kestrels

In Combinatory Logic, a Kestrel (or "K Combinator") is a function that returns a constant function, normally written `Kxy = x`. In Ruby, it might look like this:

```
# for *any* x,
kestrel.call(:foo).call(x)
  => :foo
```

Kestrels are to be found in Ruby. You may be familiar with their Ruby 1.9 name, `#tap`. Let's say you have a line like `address = Person.find(...).address` and you wish to log the person instance. With `tap`, you can inject some logging into the expression without messy temporary variables:

```
address = Person.find(...).tap { |p| logger.log "person #{p} found" }.address
```

`tap` is a method in all objects that passes `self` to a block and returns self, ignoring whatever the last item of the block happens to be. Ruby on Rails programmers will recognize the Kestrel in slightly different form:

```
address = returning Person.find(...) do |p|
  logger.log "person #{p} found"
end.address
```

Again, the result of the block is discarded, it is only there for side effects. This behaviour is the same as a Kestrel. Remember `kestrel.call(:foo).call(x)`? If I rewrite it like this, you can see the similarity:

```
Kestrel.call(:foo) do
  x
end
  => :foo
```

Both `returning` and `tap` are handy for grouping side effects together. Methods that look like this:

```
def registered_person(params = {})
  person = Person.new(params.merge(:registered => true))
  Registry.register(person)
  person.send_email_notification
  person
end
```

Can be rewritten using `returning`:

```
def registered_person(params = {})
  returning Person.new(params.merge(:registered => true)) do |person|
    Registry.register(person)
    person.send_email_notification
  end
end
```

It is obvious from the first line what will be returned and it eliminates an annoying error when the programmer neglects to make `person` the last line of the method.

2.1 Object initializer blocks

The Kestrel has also been sighted in the form of *object initializer blocks*. Consider this example using Struct[1]:

```
Contact = Struct.new(:first, :last, :email) do
  def to_hash
    Hash[*members.zip(values).flatten]
  end
end
```

The method `Struct#new` creates a new class. It also accepts an optional block, evaluating the block for side effects only. It returns the new class regardless of what happens to be in the block (it happens to evaluate the block in class scope, a small refinement).

You can use this technique when writing your own classes:

[1] http://blog.grayproductions.net/articles/all_about_struct

```
class Bird < Creature
  def initialize(*params)
    # do something with the params
    yield self if block_given?
  end
end

Forest.add(
        Bird.new(:name => 'Kestrel) { |k| combinators << k }
)
```

The pattern of wanting a Kestrel/returning/tap when you create a new object is so common that building it into object initialization is useful. And in fact, it's built into `ActiveRecord`. Methods like `new` and `create` take optional blocks, so you can write:

```
class Person < ActiveRecord::Base
  # ...
end

def registered_person(params = {})
  Person.new(params.merge(:registered => true)) do |person|
    Registry.register(person)
    person.send_email_notification
  end
end
```

In Rails, `returning` is not necessary when creating instances of your model classes, thanks to ActiveRecord's built-in object initializer blocks.

2.2 Inside, an idiomatic Ruby Kestrel

When we discussed `Struct` above, we noted that its initializer block has a slightly different behaviour than `tap` or `returning`. It takes an initializer block, but it doesn't pass the new class to the block as a parameter, it evaluates the block in the context of the new class.

Putting this into implementation terms, it evaluates the block with `self` set to the new class. This is not the same as `returning` or `tap`, both of which leave `self` untouched. We can write our own version of `returning` with the same semantics. We will call it `inside`:

```
module Kernel

  def inside(value, &block)
    value.instance_eval(&block)
    value
  end

end
```

You can use this variation on a Kestrel just like `returning`, only you do not need to specify a parameter:

```
inside [1, 2, 3] do
  uniq!
end
  => [1, 2, 3]
```

This isn't particularly noteworthy. Of more interest is your access to private methods and instance variables:

```
sna = Struct.new('Fubar') do
  attr_reader :fu
end.new

inside(sna) do
  @fu = 'bar'
end
  => #<struct Struct::Fubar >

sna.fu
  => 'bar'
```

`inside` is a Kestrel just like `returning`. No matter what value its block generates, it returns its primary argument. The only difference between the two is the evaluation environment of the block.

2.3 The Enchaining Kestrel

In **Kestrels**, we looked at `#tap` from Ruby 1.9 and `returning` from Ruby on Rails. No we'll going to look at another use for `tap`. As already explained, Ruby 1.9 includes the new method `Object#tap`. It passes the receiver to a block, then returns the receiver no matter what the block contains. The canonical example inserts some logging in the middle of a chain of method invocations:

```
address = Person.find(...).tap { |p| logger.log "person #{p} found" }.address
```

`Object#tap` is also useful when you want to execute several method on the same object without having to create a lot of temporary variables, a practice Martin Fowler calls [Method Chaining](http://martinfowler.com/dslwip/MethodChaining.html “”). Typically, you design such an object so that it returns itself in response to every modifier message. This allows you to write things like:

```
HardDrive.new.capacity(150).external.speed(7200)
```

Instead of:

```
hd = HardDrive.new
hd.capacity = 150
hd.external = true
hd.speed = 7200
```

And if you are a real fan of the Kestrel, you would design your class with an object initializer block so you could write:

```
hd = HardDrive.new do
        @capacity = 150
        @external = true
        @speed = 7200
end
```

But what do you do when handed a class that was not designed with method chaining in mind? For example, `Array#pop` returns the object being popped, not the array. Before you validate every criticism levelled against Ruby for allowing programmers to rewrite methods in core classes, consider using `#tap` with `Symbol#to_proc` or `String#to_proc` to chain methods without rewriting them.

So instead of

```
def fizz(arr)
        arr.pop
        arr.map! { |n| n * 2 }
end
```

We can write:

```
def fizz(arr)
  arr.tap(&:pop).map! { |n| n * 2 }
end
```

I often use `#tap` to enchain methods for those pesky array methods that sometimes do what you expect and sometimes don't. My most hated example is `Array#uniq!`[2]:

```
arr = [1,2,3,3,4,5]
arr.uniq, arr
        => [1,2,3,4,5], [1,2,3,3,4,5]
arr = [1,2,3,3,4,5]
arr.uniq!, arr
        => [1,2,3,4,5], [1,2,3,4,5]
arr = [1,2,3,4,5]
arr.uniq, arr
        => [1,2,3,4,5], [1,2,3,4,5]
arr = [1,2,3,4,5]
arr.uniq!, arr
        => nil, [1,2,3,4,5]
```

Let's replay that last one in slow motion:

```
[  1,  2,  3,  4,  5  ].uniq!
        => nil
```

That might be a problem. For example:

```
[1,2,3,4,5].uniq!.sort!
        => NoMethodError: undefined method `sort!' for nil:NilClass
```

`Object#tap` to the rescue: When using a method like `#uniq!` that modifies the array in place and sometimes returns the modified array but sometimes helpfully returns `nil`, I can use `#tap` to make sure I always get the array, which allows me to enchain methods:

```
[1,2,3,4,5].tap(&:uniq!).sort!
        => [1,2,3,4,5]
```

So there's another use for `#tap` (along with `Symbol#to_proc` for simple cases): We can use it when we want to enchain methods, but the methods do not return the receiver.

[2] http://ruby-doc.org/core/classes/Array.html#M002238

> In Ruby 1.9, `#tap` works exactly as described above. Ruby 1.8 does not have `#tap`, but you can obtain it by installing the andand gem. This version of `#tap` also works like a Quirky Bird, so you can write things like `HardDrive.new.tap.capacity(150)` for enchaining methods that take parameters and/or blocks. To get andand, `sudo gem install andand`. Rails users can also drop `andand.rb` in `config/initializers`.

2.4 The Obdurate Kestrel

The andand gem[3] includes `Object#tap` for Ruby 1.8. It also includes another kestrel called `#dont`. Which does what it says, or rather *doesn't* do what it says.

```
:foo.tap { p 'bar' }
bar
        => :foo # printed 'bar' before returning a value!

:foo.dont { p 'bar' }
        => :foo # without printing 'bar'!
```

`Object#dont` simply ignores the block passed to it. So what is it good for? Well, remember our logging example for `#tap`?

```
address = Person.find(...).tap { |p| logger.log "person #{p} found" }.address
```

Let's turn the logging off for a moment:

```
address = Person.find(...).dont { |p| logger.log "person #{p} found" }.address
```

And back on:

```
address = Person.find(...).tap { |p| logger.log "person #{p} found" }.address
```

I typically use it when doing certain kinds of primitive debugging. And it has another trick up its sleeve:

```
arr.dont.sort!
```

Look at that, it works with method calls like a quirky bird! So you can use it to `NOOP` methods. Now, you could have done that with `Symbol#to_proc`:

[3] http://github.com/raganwald/andand/tree

```
arr.dont(&:sort!)
```

But what about methods that take parameters and blocks?

```
JoinBetweenTwoModels.dont.create!(...) do |new_join|
        # ...
end
```

`Object#dont` is the Ruby-semantic equivalent of commenting out a method call, only it can be inserted inside of an existing expression. That's why it's called the *obdurate kestrel*. It refuses to do anything!

If you want to try `Object#dont`, or want to use `Object#tap` with Ruby 1.8, `sudo gem install andand`. Rails users can also drop `andand.rb` in `config/initializers` as mentioned above. Enjoy!

2.5 Kestrels on Rails

As mentioned, Ruby on Rails provides #returning, a method with K Combinator semantics:

```
returning(expression) do |name|
  # name is bound to the result of evaluating expression
  # this block is evaluated and the result is discarded
end
  => # the result of evaluating the expression is now returned
```

Rails also provides *object initializer blocks* for ActiveRecord models. Here's an example from one of my unit tests:

```
@board = Board.create(:dimension => 9) do |b|
  b['aa'] = 'black'
  b['bb'] = 'black'
  b['cb'] = 'black'
  b['da'] = 'black'
  b['ba'] = 'white'
  b['ca'] = 'white'
end
```

So, it looks like in Rails you can choose between an object initializer block and #returning:

```
@board = returning(Board.create(:dimension => 9)) do |b|
  b['aa'] = 'black'
  b['bb'] = 'black'
  b['cb'] = 'black'
  b['da'] = 'black'
  b['ba'] = 'white'
  b['ca'] = 'white'
end
```

In both cases the created object is returned regardless of what the block would otherwise return. But beyond that, the two Kestrels have very different semantics. "Returning" fully evaluates the expression, in this case creating the model instance in its entirety, including all of its callbacks. The object initializer block, on the other hand, is called as part of initializing the object *before* starting the lifecycle of the object including its callbacks.

"Returning" is what you want when you want to do stuff involving the fully created object and you are trying to logically group the other statements with the creation. In my case, that's what I want, I am trying to say that @board is a board with black stones on certain intersections and white stones on other intersections.

Object initialization is what you want when you want to initialize certain fields by hand and perform some calculations or logic before kicking off the object creation lifecycle. That wasn't what I wanted in this case because my `[]=` method depended on the object being initialized. So my code had a bug that was fixed when I changed from object initializers to #returning.

Summary: In Rails, object initializers are evaluated before the object's life cycle is started, #returning's block is evaluated afterwards. And that is today's *lingua obscura*.

2.6 Rewriting "Returning" in Rails

One of the most useful tools provided by Ruby on Rails is the #returning method, a simple but very useful implementation of the K Combinator or Kestrel. For example, this:

```
def registered_person(params = {})
  person = Person.new(params.merge(:registered => true))
  Registry.register(person)
  person.send_email_notification
  person
end
```

Can and should be expressed using #returning as this:

```
def registered_person(params = {})
  returning Person.new(params.merge(:registered => true)) do |person|
    Registry.register(person)
    person.send_email_notification
  end
end
```

Why? Firstly, you avoid the common bug of forgetting to return the object you are creating:

```
def broken_registered_person(params = {})
  person = Person.new(params.merge(:registered => true))
  Registry.register(person)
  person.send_email_notification
end
```

This creates the person object and does the initialization you want, but doesn't actually return it from the method, it returns whatever #send_email_notification happens to return. If you've worked hard to create fluent interfaces you might be correct by accident, but #send_email_notification could just as easily return the email it creates. Who knows?

Second, in methods like this as you read from top to bottom you are declaring what the method returns right up front:

```
def registered_person(params = {})
  returning Person.new(params.merge(:registered => true)) do # ...
    # ...
  end
end
```

It takes some optional params and returns a new person. Very clear. And the third reason I like #returning is that it logically clusters the related statements together:

```
returning Person.new(params.merge(:registered => true)) do |person|
  Registry.register(person)
  person.send_email_notification
end
```

It is very clear that these statements are all part of one logical block. As a bonus, my IDE respects that and it's easy to fold them or drag them around as a single unit. All in all, I think #returning is a big win and I even look for opportunities to refactor existing code to use it whenever I'm making changes.

DWIM

All that being said, I have observed a certain bug or misapplication of #returning from time to time. It's usually pretty subtle in production code, but I'll make it obvious with a trivial example. What does this snippet evaluate to?

```
returning [1] do |numbers|
  numbers << 2
  numbers += [3]
end
```

This is the kind of thing that sadistic interviewers use in coding quizzes. The answer is **[1, 2]**, not [1, 2, 3]. The << operator mutates the value assigned to the numbers variable, but the += statement overwrites the reference assigned to the numbers variable without changing the original value. #returning remembers the *value* originally assigned to numbers and returns it. If you have some side-effects on that value, those count. But assignment does nothing to the value.

This may seem obvious, but in my experience it is a subtle point that causes difficulty. Languages with referential transparency escape the confusion entirely, but OO languages like Ruby have this weird thing where we have to keep track of references and labels on references in our head.

Here's something contrived to look a lot more like production code. First, without #returning:

```
def working_registered_person(params = {})
  person = Person.new(params.merge(:registered => true))
  if Registry.register(person)
    person.send_email_notification
  else
    person = Person.new(:default => true)
  end
  person
end
```

And here we've refactored it to use #returning:

```
def broken_registered_person(params = {})
  returning Person.new(params.merge(:registered => true)) do |person|
    if Registry.register(person)
      person.send_email_notification
    else
      person = Person.new(:default => true)
    end
  end
end
```

Oops! This no longer works as we intended. Overwriting the `person` variable is irrelevant, #returning returns the unregistered new person no matter what. So what's going on here?

One answer is to "blame the victim." Ruby has a certain well-documented behaviour around variables and references. #returning has a certain well-documented behaviour. Any programmer who makes the above mistake is–well–mistaken. Fix the code and set the bug ticket status to Problem Between Keyboard And Chair ("PBKAC").

Another answer is to suggest that the implementation of #returning is at fault. If you write:

```
returning ... do |var|
  # ...
  var = something_else
  # ...
end
```

You intended to change what you are returning from #returning. So #returning should be changed to do what you meant. I'm on the fence about this. When folks argue that designs should cater to programmers who do not understand the ramifactions of the programming language or of the framework, I usually retort that you cannot have progress and innovation while clinging to familiarity, an argument I first heard from Jef Raskin[4]. The real meaning of "The Principle of Least Surprise" is that a design should be *internally consistent*, which is not the same thing as *familiar*.

Ruby's existing use of variables and references is certainly consistent. And once you know what #returning does, it remains consistent. However, this design decision isn't really about being consistent with Ruby's implementation, we are debating how an idiom should be designed. I think we have a blank canvas and it's reasonable to at least *consider* a version of #returning that handles assignment to the parameter.

Rewriting #returning

[4] http://weblog.raganwald.com/2008/01/programming-language-cannot-be-better.html

The RewriteRails[5] plug-in adds syntactic abstractions like Andand[6] to Rails projects without monkey-patching[7]. RewriteRails now includes its own version of #returning that overrides the #returning shipping with Rails.

When RewriteRails is processing source code, it turns code like this:

```
def registered_person(params = {})
  returning Person.new(params.merge(:registered => true)) do |person|
    if Registry.register(person)
      person.send_email_notification
    else
      person = Person.new(:default => true)
    end
  end
end
```

Into this:

```
def registered_person(params = {})
  lambda do |person|
    if Registry.register(person)
      person.send_email_notification
    else
      person = Person.new(:default => true)
    end
    person
  end.call(Person.new(params.merge(:registered => true)))
end
```

Note that in addition to turning the #returning "call" into a lambda that is invoked immediately, it also makes sure the new lambda returns the `person` variable's contents. So assignment to the variable does change what #returning appears to return.

Like all processors in RewriteRails, #returning is only rewritten in `.rr` files that you write in your project. Existing `.rb` files are not affected, including all code in the Rails framework: RewriteRails will never monkey with other people's expectations. RewriteRails doesn't physically modify the .rr files you write: The rewritten code is put in another file that the Ruby interpreter sees. So you see the code you write and RewriteRails figures out what to show the interpreter. This is a little like a Lisp macro.

[5] http://github.com/raganwald-deprecated/rewrite_rails/tree/master

[6] http://github.com/raganwald-deprecated/rewrite_rails/tree/master/doc/andand.textile

[7] http://avdi.org/devblog/2008/02/23/why-monkeypatching-is-destroying-ruby/

3 About The Author

When he's not shipping Ruby, Javascript and Java applications scaling out to millions of users, Reg "Raganwald" Braithwaite has authored libraries[1] for Javascript and Ruby programming such as Katy, JQuery Combinators, YouAreDaChef, andand, and others.

He writes about programming on his "Homoiconic[2]" un-blog as well as general-purpose ruminations on his posterous space[3]. He is also known for authoring the popular raganwald programming blog[4] from 2005-2008.

3.1 contact

Twitter: @raganwald
Email: raganwald@gmail.com

[1] http://github.com/raganwald
[2] http://github.com/raganwald/homoiconic
[3] http://raganwald.posterous.com
[4] http://weblog.raganwald.com

Reginald Braithwaite

[5] http://creativecommons.org/licenses/by-sa/2.0/deed.en