

# What Has Ruby Done For You Lately?



Adam Keys

<http://therealadam.com>

**Well?**



What has it done for you?

**Blocks**

**Well?**



What has it done for you?

# Meta-programming

**Blocks**

**Well?**



What has it done for you?

# Meta-programming

**Blocks**

Well?



**Open classes**

# Meta-programming

**Blocks**

Well?



**Syntax**

**Open classes**

# Inspiration



So all those things are great. But they came from somewhere, right?



# Lisp



Matz has called Ruby “MatzLisp” in the past. What he means is that Ruby is just enough of Lisp to make sense to mortals, while retaining as much utility as possible.

# CLOS



Matz wanted an object system. He knew it had to be simpler than CLOS, but he probably wanted as much of the power of CLOS as possible.

# Smalltalk



Ruby's methods and much of its OO-flavor is inspired by Smalltalk.

# Perl



Matz wanted Ruby to have the functionality (and utility) of Perl, but in an OO way.

# Functional Languages



Ruby's blocks were inspired by languages that included higher-order functions for operating on data in a generic way.

# A Diffusion Of Ideas



Ruby's great, but surely, there's other stuff out there. They're lurking in other languages, waiting for us to discover and use them.

# Venturing From The Comfort Of Ruby



We can explore these ideas without leaving the familiar ground of Ruby. The purists who use other languages might smirk at us, but who's keeping score?

# Programming Without Class



# We Are All Clones



Prototype-based languages do away with the distinction between potential things and actual things. Everything is a thing. Some are just more clone-ish than others.

```
napoleon = PrototypalDog.clone  
napoleon.name = "Napoleon"  
napoleon.color = "black and tan"  
napoleon.weight = 12
```



# Arbitrarily Special Objects



In Prototypical languages, anything can be different from anything else. Even if they look mostly the same. This seems chaotic, but its rather calming in practice.

```
thor = PrototypalDog.clone
thor.name = "Thor"
thor.color = "black and tan"
thor.weight = 18
thor.metaclass.send(:define_method, :arf) do
  "Will perform for food"
end
```



# Writing Ruby Without Typing Class



You can pretend to code with Prototypes in Ruby. Trying to do it without typing class is a bit of a challenge.

```
class ClassicalDachshund
  attr_accessor :name, :color, :weight

  def initialize
    name = ''
    color = ''
    weight = 0
  end

  def arf
    "Arf!"
  end

  def description
    "#{name} is a #{color} dachshund who weighs
    #{weight} pounds"
  end
end
```



```
PrototypalDog = Object.clone
class <<PrototypalDog
  attr_accessor :name, :color, :weight

  define_method(:arf) do
    "Arf!"
  end

  define_method(:description) do
    "#{name} is a #{color} dachshund who weighs
    #{weight} pounds"
  end
end
```



**“prototype.rb”**



Ara T. Howard's `prototype.rb` gem makes coding in the prototypal style slightly easier.

require 'prototype'

```
a = Prototype.new{  
  def method() 42 end  
}
```

```
b = a.clone
```

```
p a.method      #=> 42  
p b.method      #=> 42
```

```
a.extend{  
  def method2() '42' end  
}
```



# Pattern Matching



# Too Cool For Conditionals



One particularly entertaining way to annoy your co-workers is to code without using if/else statements. Traditionally, your best friend in this quest is polymorphism.

```
if heat > 20
  case gizmo
  when 'honk'
    :flop
  when 'flurf'
    :blurp
  when 14
    :honk
  else
    true
  end
elsif heat < 20 && gate?
  if zonks? || flerp > 20
    'zounds!'
  else
    'drat'
  end
end
end
```



Quick, find the bug!

# Let The Language Guys Write The Code That Breaks



Most bugs arise because you take the wrong path down a conditional. So why not push all that nastiness into the compiler or runtime?

# Pattern Matching On Parameters



# Topher Cyll's "multi"



```
multi(:fac, 0) { 1 }
```

```
multi(:fac, Integer) { |x| x * fac(x - 1) }
```

```
fac(5) # => 120
```



```
multi(:fac, 0) { 1 }
multi(:fac, Integer) { |x| x * fac(x - 1) }

fac(5) # => 120

multi(:scales?, lambda { |x| x > 100_000 })
{ true }
multi(:scales?, Object) { false }

scales?(100) # => false
scales?(100_000_000) # => true
```



# Pattern Matching On Data



```
require 'rubygems'  
require 'smulti'
```

```
keyword = 'fungdark'
```

```
smulti(:authenticate, 'fungdark') { true }  
smulti(:authenticate, /./) do |_, rest|  
  authenticate(rest)  
end  
smulti(:authenticate, //) { false }
```

```
authenticate('You are a fungdark') # => true  
authenticate('My voice is my password, verify me')  
# => false
```



# *Lazy* Programming



# Functions On Functions



You've been doing higher-order programming in Ruby the whole time. Doing it with a purpose is even more fun!

```
Dog = Struct.new(:name, :size, :color)
```

```
dogs = [Dog.new('Napoleon', :tweenie, :black_and_tan),  
        Dog.new('Thor', :standard, :black_and_tan),  
        Dog.new('Fred', :tweenie, :red_dapple)]
```



```
Dog = Struct.new(:name, :size, :color)
```

```
dogs = [Dog.new('Napoleon', :tweenie, :black_and_tan),  
        Dog.new('Thor', :standard, :black_and_tan),  
        Dog.new('Fred', :tweenie, :red_dapple)]
```

```
dogs.select { |d|  
  d.name == 'Napoleon'  
}.map { |d|  
  d.name  
} # => ["Napoleon"]
```



```
Dog = Struct.new(:name, :size, :color)
```

```
dogs = [Dog.new('Napoleon', :tweenie, :black_and_tan),  
        Dog.new('Thor', :standard, :black_and_tan),  
        Dog.new('Fred', :tweenie, :red_dapple)]
```


```
dogs.partition { |d|  
  d.size == :tweenie  
}.map { |sizes|  
  sizes.map { |d|  
    d.name  
  } } # => [[ "Napoleon", "Fred"],  
["Thor"]]
```



# Delaying Execution With Thunks



There are times when you know **what** you need to do, but not **when** to do it. Thunks, despite the hilarious name, are what you need.



```
def slow_add(x, y)
  sleep 10
  x + y
end
```



```
def slow_add(x, y)
  sleep 10
  x + y
end
```

```
Time.now # => Sun Jul 20 16:07:53 -0500 2008
slow_add(2, 2)
Time.now # => Sun Jul 20 16:08:03 -0500 2008
```



```
def slow_add(x, y)
  sleep 10
  x + y
end
```

```
Time.now # => Sun Jul 20 16:07:53 -0500 2008
slow_add(2, 2)
Time.now # => Sun Jul 20 16:08:03 -0500 2008
```

```
foo = lambda { slow_add(2, 2) }
Time.now # => Sun Jul 20 16:08:03 -0500 2008
foo.call
Time.now # => Sun Jul 20 16:08:13 -0500 2008
```



# Patiently Evaluating The Patient



Lazy evaluation lets us do all sorts of neat tricks with computation. It also yields cool tools like defining our own syntax and just-in-time calculation.

```
myif = lambda do |cond, if0, else0|  
  cond ? if0 : else0  
end
```

```
awesome = myif[1 == 1,  
              "ruby is awesome",  
              "ruby is lame"] # => "ruby is awesome"
```



```
myif = lambda do |cond, if0, else0|  
  cond ? if0 : else0  
end
```

```
awesome = myif[1 == 1,  
              "ruby is awesome",  
              "ruby is lame"] # => "ruby is awesome"
```

```
trickierif = lambda do |cond, if0, else0|  
  cond[] ? if0[] : else0[]  
end
```

```
trickierif[lambda { "ruby is awesome" == "ruby is  
awesome" },  
          lambda { "Ruby rules" },  
          lambda { "Ruby is lame-esque" }]
```



```
Book = Struct.new(:title, :pages)
```

```
gof = Book.new('Design Patterns', 400)  
dragon = Book.new('Principles of Compiler Design', 800)  
wizard = Book.new('SICP', 900)
```

```
class Array  
  def hardcore_books(&block)  
    self.inject([]) do |ary, book|  
      block.call(book) ? ary << book : ary  
    end  
  end  
end
```

```
books = [gof, dragon, wizard]  
awesome_books = books.hardcore_books &lambda { |b|  
  b.pages > 400  
}
```



# We Could All Do With Fewer Arguments



Currying lets us peel arguments off a function. That makes it easier to call from other places, or lets us do away with costly calculations.

def adder(x, y)  
 x + y  
end

incr = lambda { |x| adder(x, 1) }

incr[1] # => 2



# Zen And The Art Of Programming Without Side-effects



The coolest tricks in functional programming come when we discipline ourselves to write functions which change nothing. This makes the wizards writing the compilers happy, which makes us happy because then we don't have to think about them.

# Your Brain On Monads



Monads are the zen trick that lets us write useful programs, but still pretend we're not invoking side-effects. Even if you never use them, its fun to just rewire your brain by trying to grok them.

# Parting Thoughts



# Embrace Your Inner Novelty Junky



Languages are neat. The more you know, the more you can learn. Its fun to impress your friends.

# The Only Way To Learn Is By Falling Down



Not all ideas come out gracefully in Ruby. Every time you fail to grok a language, you learn something about it **and** yourself. Experimenting is the most important part.

# Inspiration



When I started this talk, I was thinking I'd show tidbits of these languages and how they inspired the various concepts we covered. But, our time was short, so I just extracted their essence. Nevertheless, I strongly encourage you to check out one or more of these



**Io**

**Inspiration**



When I started this talk, I was thinking I'd show tidbits of these languages and how they inspired the various concepts we covered. But, our time was short, so I just extracted their essence. Nevertheless, I strongly encourage you to check out one or more of these



**Io**

**Inspiration**



**Erlang**



When I started this talk, I was thinking I'd show tidbits of these languages and how they inspired the various concepts we covered. But, our time was short, so I just extracted their essence. Nevertheless, I strongly encourage you to check out one or more of these



# Haskell

# Io

# Inspiration



# Erlang



When I started this talk, I was thinking I'd show tidbits of these languages and how they inspired the various concepts we covered. But, our time was short, so I just extracted their essence. Nevertheless, I strongly encourage you to check out one or more of these

**Thanks**



<http://therealadam.com>