Philly on Rails

FEBRUARY 2007

Ruby 101

PhillyOnRails Wackos

IRb - Interactive Ruby

```
meara@wildfire:~> irb
irb(main):001:0> puts 'Hello, World'
Hello, World
=> nil
irb(main):002:0> class Hello
irb(main):003:1>   def greet
irb(main):004:2>     puts 'Hello World'
irb(main):005:2>   end
irb(main):006:1> end
=> nil
irb(main):007:0> Hello.new.greet
Hello World
=> nil
irb(main):008:0>
```


Meet Ruby...

- Object-oriented with single inheritance
  - Almost everything is an object
  - No interfaces or abstract classes
  - Classes are open, and code can be written outside of them (ends up in Kernel)
  - Modules for namespacing and mixins
- Strongly, dynamically typed
- Allows operator overloading
- Supports closures
- Provides garbage collection

General Flavor

```
require 'yaml'

module ActiveRecord

  class AttributeAssignmentError < ActiveRecordError

    attr_reader :exception, :attribute

    def initialize(message, exception, attribute)
      @exception = exception
      @attribute = attribute
      @message = message
    end

  end

end
```


Variables and Identifiers

- No variable declarations; variables spring into existence when you use them
- Local variable names start with lowercase letter or underscore
- Instance variable names start with @
- Class variable names start with @@
- Method names start with lowercase letter or underscore
- Class names and constants start with capital letter

Variables and Identifiers

```
class Television

    attr_accessor :inches, :current_show

    def initialize(inches, current_show)
      @inches = inches
      @current_show = current_show
    end

    def watch_show(show)
      show_name = show.title.capitalize
      @current_show = show_name
      @@all_shows ||= []
      @@all_shows << show_name
    end
end
```


Comments

- # comments everything until end of line
- Use =begin and =end for longer, inline documentation

```
# This is a basic comment
my_array = [1, 2, 'blue']
my_array << 'red'  # Here's another comment
puts my_array[3]

=begin
Here's a really long comment.
It spans more than one line.
Without having to put # at the beginning of each
=end
```


Strings

- Use single quotes for minimal interpretation
- Use double quotes to allow substitution
- #{expression} to embed an expression
- Leave off curly braces for simple output of instance, class and global variables

```
name = 'John'

blog_title = “#{name}'s Blog”
blog_url = “#@some_instance_var/#{name.downcase}/blog”
blog_log = “#$some_global/file.log”
```


Arrays and Hashes

- Arrays can be indexed or associative
- Associative arrays are called hashes
- Convenient syntax for:
  - appending to arrays
  - populating and accessing hashes

```
my_array = [1, 2, 'blue']
my_array << 'red'
puts my_array[3]

my_hash = { :title => 'blue', :number => 1 }
puts my_hash[:title]
```


Branching

- Basic **if, elsif, else, end** sequence
- Can invert with **unless**
- Both can be placed at the end of a statement instead of at the beginning of a block
- **case** statements start with **case** keyword, followed by **when** clauses and then end
- **when** clauses support regular expressions, ranges and more

Branching

```
if kitchen.hot?
  get_out
elsif kitchen.cold?
  stir_up_trouble
else
  look_busy
end

stir_up_trouble unless kitchen.hot?

case kitchen.temperature
  when -10...0
    puts 'Brrrr'
  when 0...30
    puts 'Mmmmm'
  else
    puts 'Ouch!'
end
```


Looping

- Supports **for**, **while** and **until** loops
- Can be used as modifiers at the end of a statement, not just at the beginning of a block
- **for** loops use **.each** iterators under the covers
- **.each{|item| do_something }** is fairly common
- Use indexed for loop or **.each_index** if you need a loop variable

Looping

```
for x in list do
  #something
end

while !done? do
  #something
end

puts '.' while still_waiting?

list.each do |item|
  puts item
end

sleep 5 until done?
```


Defining classes and methods

- Use **class**, **def** and **<** (extends)
- Define **initialize** method (constructor)
- Don't use **do** keyword
- Classes can have code outside of methods

```
class FourPaneWindow < Window
  panes 4
  def initialize (tint_color)
    @color = tint_color
    order_new_panes
    remove_caulk
    insert_pane
    caulk
  end
end
```


Method Punctuation

- Methods can end in a symbol
- ? is often used for boolean return values
- ! is often used for destructive methods
- Other symbols used for operator overload

```
def open_for_business?
  during_business_hours? and staffed?
end

def name=(new_name)
  @name = new_name
  @name.capitalize!
end
```


Exceptions

- Use **raise** or **fail** to raise an exception and begin,rescue,else,ensure,end to handle one or more

```
def negotiate_world_peace(balloon_quantity)
  raise NotYetImplementedError
end
begin
  negotiate_world_peace gets.chomp
rescue NotYetImplementedError
  puts 'guess we will have to wait on that one'
rescue ArgumentError
  puts 'balloon quantity must be a number'
else
  puts 'something went wrong'
ensure
  pop_balloons
end
```


Duck Typing

- Can use **is_a?** or **kind_of?** to test type
- But better to use **responds_to?**

```
if product.kind_of? Generator
  product.generate_power
else
  backup.generate_power
end

if product.responds_to? :generate_power
  product.generate_power
else
  backup_generator.generate_power
end
```


Public, private, protected

- Place these keywords within class definition to mark certain methods as public, private or protected
  - If you give it a symbol as a parameter, it will only apply to that method
  - If you don't give it a parameter, it will apply to all methods afterwards until the next access modifier

nil, false and 0

- nil and false evaluate to false
- 0 evalutes to true
- All of these are objects

```
nil.class     # => NilClass

false.class   # => FalseClass

0.class       # => Fixnum

if nil or false
  # this won't happen
elsif 0
  # this will happen
end
```


Require vs. Include

- **require** ensures that a file is loaded
- **include** mixes it into the current class
- **extend** lets you mix in a module to a single object

```
require 'yaml'
class ShoppingList
  include Enumerable
  def each; some_code; end
  def <=>(other); some_code; end
end

s = ShoppingList.new
if s.include?('eggs')
...
```


Inspecting Objects

- Use Kernel.p method for quick and easy object dumps

```
class Product
  attr_accessor :name, :description
end

piano = Product.new
piano.name = 'Piano'
piano.description = '88 keys'

p piano

# Prints #<Product:0x401fe4cc @description="88 keys",
#                             @name="Piano">
```


Executing system commands

- Surround in backticks or **%x[...]** to capture output
- Call **system** to get boolean success value
- Call **exec** to have it take over process
- All of these are implemented in **Kernel**

```
directory_listing = `ls -l`

if (system “rpm -ivh ${my_file}”)
  puts 'success'
else
  exec('poweroff')
end
```


Blocks, Procs and lambdas

- Also called “closures”
- Block of code that remembers its context
- Commonly used to:
  - Allow library to maintain control, cleanup its own resources after the block has executed
  - Implement iterators
  - Register callbacks
- Can convert to object with Proc.new or Kernel.lambda
- Lambdas get their own return scope

Blocks, Procs and lambdas

```ruby
File.open('README.txt', 'r') do |file|
 # do something
end

my_array = [1, 6, 1, 7, 8]

# Tradition to use curly braces for single-line blocks
my_array.each { |num|  puts num**2 }

def batter_up
  yield 'strike'
  yield 'strike'
  yield 'hit'
end

batter_up { |result| puts result.upcase+'!' }
```


Other Rubyisms

- Ranges
- aliasing methods
- $0 == __FILE__
- redo, retry
- freeze

Gotcha!

- No increment operators (e.g. ++, --)
- **and**, **or** lower precedence than **&&**, **||**
- "else if" is written **elsif**
- 0 is not false. Only **nil** and **false** are.
- **case** adds **break** at end of each **when**
- Ranges are not interchangeable with arrays. You need to call **to_a** first.
- Forgetting @ on an instance variable creates local variable with same name

Gotcha!

- Spacing can be significant, use spaces around symbols unless you mean not to
- `+=`, `*=` and other such operators are defined magically in terms of `=`, so if you want to override `foo += 10`, you'll need to override `foo=()`, not `foo+=()`
- `+` performs addition on numbers and concatenation on strings
- `my_string + my_num` doesn't work, use `my_string = my_num.to_s`

Less is More

- Parentheses are mostly optional
- Last expression in a method or block is the return value
- You can leave off the curly braces on a hash that is the last (or only) argument
- Multiple assignment without temp variables (e.g. `x, y = y, x`)
- `attr_accessor`, `attr_reader` and `attr_writer` generate access methods for instance variables

Less is More

```
class Square
  attr_accessor :height, :width
  attr_reader :labels

  def initialize
    @labels = {}
  end

  def area
    height * width
  end

  def add_labels(map)
    @labels.merge!(map)
  end
end

s = Square.new
s.height, s.width = 100, 90
s.add_labels(:name => 'box', :price => '50')
```


Classes are open

- Add new methods
- Override existing methods
- Inject dependencies or config at runtime
- Conditionally define methods at runtime

```
class String
  def one_letter?
    length == 1
  end
end

puts 'ruby'.one_letter?
puts 'r'.one_letter?
```


Definitions are executed

- You can conditionally define a method
- During execution of a class definition, self refers to the class itself

```
class Widget

  if $config.short_process
    def manufacture ; make; sell; end
  else
    def manufacture
      make
      paint
      price
      sell
    end
  end
end
```


Variables have no type

- They are just pointers to objects
- They spring into existence when you first use them
- Can be reused across object types

```
some_var = “hello”
some_var = 1
some_var = 19.class
```


method_missing

- Called whenever a called method doesn't exist
- Often used to implement DSLs or convenience “builder” classes

```
class Parrot
  def method_missing(name)
    puts name
  end
end

p = Parrot.new
p.hello
p.wanna_cracker?
p.bye
```


Classes are objects

- They can be assigned to variables
- They can have their own instance variables

```
apple = 'banana'.class
coconut = apple.new('coconut')

class Fruit
  # this is a class variable, not an instance variable
  @colors = ['red', 'green', 'orange']
  def eat
    # this is an instance variable
    @eaten = true
  end
end
```


<=> (spaceship) operator

- Compares two values and returns -1, 0, 1
- Makes sorting blocks very compact
- Override to specify how your objects should be compared

```
products.sort! { |p1, p2| p1.price <=> p2.price }

# or we could do this

class Product
  def <=>(o)
    self.price <=> o.price
  end
end
```


* (splat) operator

- Also called “star” or “unary unarray”
- Explodes array into list of piped values
- Not necessary for basic multiple assignment (e.g. x,y,z = [1,2,3])
- Useful for:
  - passing in an array to a method that expects list of args
  - collecting method parameters into an array
  - shorthand in a when clause for matching any value within the splatted array

* (splat) operator

```
def foo(x, y, z)
  puts “#{x} #{y} #{z}”
end

my_array = [1, 2, 3]
foo my_array # BAD CODE
foo *my_array # GOOD CODE

my_num = 3

case my_num
  when *my_array
    puts 'low num'
  when 4, 5
    puts 'high num'
  else
    puts 'out of bounds'
end

def bar(*args) puts args.join(“, “) end
```


=== (threequal) operator

- Also called “case equality”
- Used to evaluate matches to when clauses in case statements
- Default implementations do reasonable things (e.g. Range.=== checks if the argument is in its range)
- Tweak your === method definition if you want objects of that class to have special behavior within a when clause

=== (threequal) operator

```
class Recipe
  def ===(val)
     @ingredients.include? val
  end
end

pancakes = Recipe.new  # imagine adding ingredients next
fruitcake = Recipe.new # imagine adding ingredients next

case 'milk'
  when pancakes,fruitcake
    puts 'we need milk'
  else
    puts 'we don't need milk'
end
```


Eigenclasses

- Also called “Singletons” or “Metaclasses”
- Referred to as “Object Specialization”
- Add functionality to a specific instance without adding it to the whole class
- Classes are objects, so they have their own eigenclasses. Use these to:
  - Conveniently define class methods
  - Combine with inheritance to implement beautiful DSLs like ActiveRecord

Eigenclasses (1)

```
special_event = Event.new

class << special_event

  def priority
    if near_deadline?
      1000
    else
      10
    end
  end

end
```


Eigenclasses (2)

```
class Recipe
  # one way of doing it
  def self.list_possible_ingredients
     “bread, milk, butter”
  end
  def self.list_possible_units
     “pounds, cups, grams”
  end

  # another way
  class << self
    def list_possible_ingredients
      “bread, milk, butter”
    end
    def list_possible_units
      “pounds, cups, grams”
    end
  end
end
```


Eigenclasses (3)

```
class Object
  def metaclass; class << self; self; end; end
  def meta_eval &blk; metaclass.instance_eval &blk; end
  def meta_def name, &blk
     meta_eval { define_method name, &blk }
  end
end

class Recipe
  def self.temperature(degrees)
    meta_def :temperature do; degrees; end
  end
end

class Muffins < Recipe
  temperature 500
end

Muffins.temperature # => 500
```