Web development with Ruby on Rails

Unit 3: Ruby

Announcements

Office Hours

Mat

Wednesdays 9am - 12pm

Innovation Station #2, Weiss Tech House

Jennifer

Fridays 4p-5p

2nd Floor of Houston Hall

David

Thursdays, 8pm-9:30pm, Moore 100A

Sundays, 11pm-12am, Dubois Computer Lab

IRC

matschaffer

irc.freenode.net #philly.rb

Sub on 10/3

Ruby

- Designed by Yukihiro Matsumoto (“Matz”)
- Interpreted OO language
- Sometimes described as a cross between Smalltalk and Perl

Ruby Features

- Concise but readable
- Pure OO (no primitives)
- Dynamic typing (“Duck typing”)
- Functional programming (blocks/closures)
- Metaprogramming

Ruby Intangibles

- Thriving ecosystem
- Maximizes developer “happiness”
- LOTS of discussion and opinion on this topic, e.g. this thread on Stack Overflow
  `http://bit.ly/elegantruby`

Many Rubies

- Matz's Ruby Interpreter (MRI)
- JRuby
- Rubinius
- MacRuby
- MagLev
- mRuby

Just Enough Ruby

Syntax

Statements

```
# Separated by line breaks

statement_one
statement_two
```

```
# Can use semicolon to put several
# statements on one line

statement_one; statement_two
```

```
# Operator, comma or dot can be
# followed by a line break

my_name = "Mat " +
          "Schaffer"
```

```
# Strings can contain
# line breaks

my_name = "Mat
Schaffer"
```

Data Types - No Primitives

Constants, classes and modules

```
PI
MyClass
MyModule
```

Symbols

```
:purple
```

Strings

```
"purple"
```

Data Types (cont)

Arrays

```
["one", 2, :four]
```

Hashes 1.8

```
{ :good => “four legs”,
  :bad => “two legs” }
```

Hashes 1.9

```
{good:"four legs", bad:"two legs"}
```

Numbers

```
100
3.14159
1_000_000
```

Strings

- Characters quoted by single-quotes (') or double-quotes (")

Syntax

```
"This is a String"
```

Inline Ruby interpretation

- Double-quotes interpolate, single-quotes do not.

Example

```
irb> name = "Smith"
irb> "Hello, #{name}!"

=> Hello, Smith!
```

Symbols

- Like an immutable string
- More efficient, use single memory address
- Not interchangeable

String

```
"blue"
```

Symbol

```
:blue
```

Jim Weirich says...

(on strings vs. symbols)

- If the contents of the object are important, use a String
- If the identity of the object is important, use a Symbol

Classes

Defining / Instantiating

```ruby
class Person
  def initialize(name)
    @name = name
  end
  def greet
    puts "#{@name} says hi"
  end
end

ted = Person.new("Ted")
ted.greet
```

Variable Scope / Naming

```
CONSTANT
AnotherConstant
$global_variable
local_variable
@instance_variable
@@class_variable
```

Inheritance

Single inheritance only

```
class Car < Vehicle
...
end
```

- `Car` "is a kind of" `Vehicle`
- Can re-open and extend/redefine classes
- Use Modules for "multiple inheritance"

Modules

- Modules model qualities and abilities of things
- Cannot be instantiated directly
- Class names are typically nouns, whereas module names are typically adjectives

Module Example

```
module Driveable
  def drive(speed=55)
    puts "Zoom, going #{speed} mph"
  end
end

class Car < Vehicle
  include Driveable
end
```

Enumerable

- Commonly used and powerful module
- Adds traversal and searching methods to collections
- Methods include:

```
collect, find, inject, map, member?, reject, select, sort,
                        sort_by, to_a
```

Methods

- Method calls are really messages sent to the receiver

Equivalent

```
bar.foo
bar.send(:foo)
```

Parentheses are optional

You may skip parentheses around method parameters

```
bar.foo(6)
bar.foo 6
```

Class / Instance Methods

```ruby
class Bar
  def self.method1
    puts "This is a class method"
  end
end

Bar.method1
```

```ruby
class Bar
  class << self
    def method2
      puts "This is also a class method"
    end
  end
end
Bar.method2
```

```ruby
class Bar
  def method3
    puts "This is an instance method"
  end
end
Bar.new.method3
```

Calling other methods

Self is implied

```
class TwoMethods
  def bar
    puts "someone called bar"
  end
  def foo
    puts "gonna call bar"
    bar
  end
end
```

But can also be explicit

```
class TwoMethods
  def bar
    puts "someone called bar"
  end
  def foo
    puts "gonna call bar"
    bar = 1234
    self.bar
  end
end
```

Public, Protected, Private

- Public - `thing.foo`, `self.foo` or `foo`
- Protected - `self.foo` or `foo`
- Private - only `foo`
- (not like Java, best to avoid for now)

Open Classes

```
def Hash.method1
  puts "This is a class method"
end
Hash.method1    # => "...class..."

class String
  def method5
    puts "This is an instance method"
  end
end
"abc".method5 # => "...instance..."
```

Return Values

- Last expression evaluated in a method is the return value
- All methods have a return value (although it might be nil)

Example

```
def square(val)
  val*val
end

irb> square 4
=> 16
```

Control Flow

```
num = 42
if num > 0
   puts "POSITIVE"
elsif(num == 0)
   puts "ZERO"
else
   puts "NEGATIVE"
end
```

Control Flow

```
num = 42

puts "POSITIVE" if num > 0
```

Case Statements

```
num = 10
case num
when 1
   puts "ONE"
when 10..20
   puts "BETWEEN 10 AND 20"
when 40, 60
   puts "FORTY OR SIXTY"
else
   puts "UNSUPPORTED VALUE"
end
```

Case Statements

```
word = "ONE"
foo = case word
      when "ONE"
         1
      when "TWO"
         2
      when "THREE"
         3
      else
         42
      end
```

Exceptions

rescuing them

```
begin
  File.open('non-existent')
rescue Errno::ENOENT
  puts "File didn't exist"
end
```

Raising them

```
if a != b
  raise "A wasn't equal to b!"
end
```

or an exception class

```
class NotEqualError < StandardError; end

if a != b
  raise NotEqualError.new
end
```

Can also retry

```
tries = 0
begin
  tries += 1
  some_flakey_api
rescue HTTPError
  retry unless tries >= 5
end
```

Or clean up with ensure

```
begin
  output_file = File.open('output.txt')
  something_bad_with(output_file)
rescue SomethingBad
  puts "it happened again"
ensure
  output_file.close
end
```

Trying stuff out: irb

```
> irb
1.9.2-p290 :001 > 1 + 1
 => 2
```

Trying stuff out: rails console

```
> rails c
Loading development environment (Rails
1.9.3p194 :001 > Pledge.new
 => #<Pledge id: nil ...>
```

Default Arguments

Example

```
def foo(text="Hi!")
  puts text
end

foo
=> Hi!
```

*args

```ruby
def foo(*args)
  # print the first and last arguments
  puts args[0]
  puts args[args.length-1]
end
```

Example

```ruby
foo(1,2,3)
=> 1
=> 3
```

Duck Typing

Query an object to find out what it does

```
if duck.respond_to?(:quack)
    puts "It quacks!"
end
```

Blocks, Procs & Lambdas

- Groupings of code that can be passed around and executed by other code.

Blocks

```
{ puts "This is a block." }
```

```
do
  puts "This is also a block"
end
```

Blocks with Arguments

```
{ |arg| puts "The arg is #{arg}" }
```

```
do |arg|
  puts "The arg to is #{arg}"
end
```

Invoking Block Logic

Example

```ruby
def my_method(arg)
  puts "before block invocation"
  yield arg
  puts "after block invocation"
end
```

Example

```ruby
my_method(42) do |arg|
  puts "The arg is #{arg}"
end
=> before block invocation
=> The arg is 42
=> after block invocation
```

Blocks

```
# in fib.rb:
def fibonacci(maximum)
  n1, n2 = 1, 1
  while n1 <= maximum
    yield n1
    n1, n2 = n2, n1+n2
  end
end

fibonacci(100) { |n| print n, " " }
```

```
$ ruby fib.rb
1 1 2 3 5 8 13 21 34 55 89
```

Blocks and Enumerable Methods

```
[1,2,3].each {|num| print num, " "}
=> 1 2 3

[1,2,3].each do |num|
  puts "#{num}"
  puts "#{num + num}"
end
=> 1
=> 2
...
```

Blocks and Enumerable Methods

```
[1,2,3].map{|num| num * num}
=> [1,4,9]
```

Enumerable Methods and Blocks

```
[1,2,3].inject(0) do |sum,num|
    puts "#{sum} #{num}"
    sum + num
end
0 1
1 2
3 3
=> 6
```

Procs

```
my_proc = Proc.new { |n| n+1 }
my_proc.call 41
=> 42
```

Block to Proc Conversion

```ruby
def my_method(arg, &block)
    puts "before block invocation"
    block.call arg
    puts "after block invocation"
end
```

Example

```ruby
my_method(42) do |arg|
  puts "The arg to this block is #{arg}"
end
```

Proc to Block Conversion

```
my_proc = Proc.new do |arg|
     "Arg is #{arg}"}
end
[1,2,3].each(&my_proc)
```

Lambda

```ruby
my_lambda = lambda   { |x,y| puts x*y
my_proc   = Proc.new { |x,y| puts x*y

# Prints 6 as expected
my_lambda.call(2,3)

# Fails with an ArgumentError
my_lambda.call(2,3,4)

# What does this return?
my_proc.call(2,3,4)
```

Ruby 1.9 Lambda Syntax

1.8 & 1.9

```
my_lambda = lambda { |x,y| puts x*y }
my_lamba.call(x,y)
my_lambda[x,y]
```

1.9

```
my_lambda = -> x,y { puts x*y }
my_lambda.(x,y)
```

Block Argument Scope

1.8

```
x=1
[1,2].each {|x| x + 40}
x
=> 42
```

1.9

```
x=1
[1,2].each {|x| x + 40}
x
=> 1
```

Ruby's Expressivness

```
num = 1
puts "POSITIVE" unless num <= 0
=> POSITIVE
puts "POSITIVE" if num > 0
=> POSITIVE
```

Indicating Destructiveness

```
str = "abc"
loud_str = str.upcase
loud_str
=> "ABC"

str
=> "abc"
str.upcase!
str
=> "ABC"
```

Further reading

rubykoans.com

mislav.uniqpath.com/poignant-guide

Next week:

Rails Requests and Data

Homework: Ruby Lab

Submit parts 4 and 6