RAD Lab
UC Berkeley

Ruby on Rails Short Course: Just Enough Ruby

William Sobel

UC Berkeley RAD Lab

Outline of the day

1. Web apps, MVC, SQL, Hello World
2. Just enough Ruby
3. Basic Rails

Lunch break

4. Advanced model relations
5. AJAX & intro to testing
6. Configure and Deploy

Informal discussion: RoR and pedagogy

Section 2

- Overview of the Language
- Conventions
- Classes
- Closures
- Iteration
- Modules
- Enumerations
- Meta-Programming
- Advanced
- Are you kidding!

Why Ruby?

- Purely Object-Oriented
  - Everything’s an Object
- Focus on Developer Productivity
- Borrows from:
  - Lisp, Perl, Smalltalk, and CLU
- Consistent Syntax
  - Easy to Learn

Philosophy

“Often people, especially computer engineers, focus on the machines. They think, "By doing this, the machine will run faster. By doing this, the machine will run more effectively. By doing this, the machine will something something something." They are focusing on machines. But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines. We are the masters. They are the slaves.”

Yukihiro "Matz" Matsumoto

Scoping

Variable names indicate **scope, not type**

- In perl @ is an array and % is a hash
- In Ruby the leading @ indicates instance variable

Conventions

`foo _bar _123`	Local Variables
`@foo @trucks @_123 @Cat`	Instance Variables
`@@foo @@car @@dag`	Class Variables
`Cow COW`	Constants
`$rails $a $_123 $Horse`	Global Variables
`Math::PI`	Module Constants
`1 2 3_000 1_234_244_432_444`	Integers and Bignums
`1.0 2.0`	Floating Point Numbers
`Math::sin(x)`	Module Functions
`:symbol :”A-Symbol”`	Symbols (Singletons)
`/^[Rr]egexp/`	Regular Expressions
`[1, “1”, :one]`	Array
`{ :one => 1, :two => 2}`	Hash

Variables

- Any variable can hold any type, everything's an object
- Always pass-by-reference
- The only immutable types are numbers
  - A 1 cannot become a 2 no matter how nicely you ask
  - If you add 1 to 1 (1 + 1), you have not changed the 1 into a 2, you've created a new 2 object

Variable Creation

- Variables are created dynamically

```
# Creates a variable 'a' and bind the String 'fred'
a = 'Fred'

# Bind the String to an instance variable @a, there
# is no relationship between a and @a
@a = 'Bob'

# Binds the class variable to the instance variable
# @a
@@a = @a

@@a
⇒ 'Bob'

@@a << ' Smith'
⇒ 'Bob Smith'
@a
=> 'Bob Smith'
```

Parallel Assignment

```
a = 1
b = 2
a, b = b, a
a => 2
b => 1
def foo(x); [x, x+1]; end
a, b = foo(4)
a => 4, b => 5

# Easy way of slicing the first element
a, = [6, 7, 8]
a => 6
```

Methods

```
# Methods
class Account
  def initialize
    @balance = 0
  end
  def deposit(amount)
    @balance += amount
  end
  def withdraw(amount)
    @balance -= amount
  end

  def method1(name, *rest); …; end
  def method2(name, &block); …; end

  # method3(:fred, :shoe_size => 11, :height => 70)
  def method3(name, options  = {}); …; end
  def method4(name, show_size = 11, height = 70)

  # A class method
  def self.connection
    @@connection
  end
end
```

Method Protection

- Ruby has three levels of protection
  - **Public** - can be called by anyone
  - **Protected** - can only be called by any instance of the object and subclasses
  - **Private** - cannot be called with a specific receiver (must use self)

```
class Account
    def initialize; …; end
  protected
    def transfer(xxx); …; end
  public
    def balance; …; end
  def update(x); …; end
  protected :update
end
```

Method Conventions

- Methods ending in '?' return boolean results
  - **`empty?, zero?, include?, eql?, member?, success?, stoped?, any?, all?,`** …
- Methods ending in '!' are possibly dangerous
  - Used to distinguish methods where one version modifies the receiver. **`map!, sort!, reject!,`** …
  - Not all methods that modify the receiver use !. **`delete, delete_at`**

```
a = [1, 2, 3]
a.map { |i| i + 1 } # Returns a new array
=> [2, 3, 4]
a => [1, 2, 3]

a.map! { |i| i + 1 }
=> [2, 3, 4]
a => [2, 3, 4]
```

Classes

```
# Classes are easy to define
class TheClass
  def initialize(a) # Called from TheClass.new
    @value = a
  end
end

# Creating a subclass
class SubClass < TheClass
  def initialize(a)
    @b = 200
    super(a + b) # Calls superclass method
  end
end
```

Mutability

```
# All classes can be modified
class Fixnum
  def +(x)
    self * x
  end
end

1 + 5
⇒ 5

12 + 4
⇒ 48
```

Now For Something Useful

```
# Before
irb(main):001:0> nil + 1
NoMethodError: undefined method `+' for nil:NilClass
        from (irb):1

# Defined
class NilClass
  def method_missing(*args)
    nil
  end
end

# After
irb(main):007:0> nil + 1
=> nil
```

Operators

```
# Methods can be created dynamically
class Foo
  def a; @a; end
end

# Operators are methods as well
class Foo
  def a=(b); @a = b; end
  def a+(b); @a += b; end
end

f = Foo.new
f.a = 1
f.a += 1
f.a
=> 2
```

Closures & Yield

```
# Yield calls the block supplied
def twice
  yield
  yield
end

twice { puts 'Get your shoes on…' }

Get your shoes on…
Get your shoes on…

# Closures are lexically scoped
name = 'Julien'
twice { puts "#{name}, Get your shoes on!" }

Julien, Get your shoes on!
Julien, Get your shoes on!
```

Iteration

```
# Never needed in Ruby…
for (i = 0; i < list.length; i++) {
  x = list[i];
  // do something with x
}

# Do this
list.each do |x|
  # do something with x
end

# If you really want the index
list.each_with_index do |x, i|
  # do something with x
end
```

Iteration (p2)

```
# Get the 5 objects after the first
int last = list.length < 6 ? list.length : 6
for (int i = 1; i < last; i++) {
  x = list[i];
  // do something with x
}

# In ruby
list[1,5].each do |x|
  # do something with x
end
```

Regular Expressions

```
'string: "hello \"bob\"!"' =~ /"([^"\\]*(\\.[^"\\"]*)*)"/
⇒ 8

$1
=> "hello \"bob\"!"

# Regular expressions are Objects:
m = /([A-Z]+)/i.match(“123 xxx 456”)
⇒ #<MatchData:0x82a4c>
m.to_a
⇒ [‘xxx’, ‘xxx’]
m.methods

=>  ["==", "===", "=~", "[]", "__id__", "__send__", "begin", "captures",
   "class", "clone", "display", "dup", "end", "eql?", "equal?", "extend",
   "freeze", "frozen?", "hash", "id", "inspect", "instance_eval",
   "instance_of?", …, "to_a", "to_s", "type", "untaint", "values_at"]
```

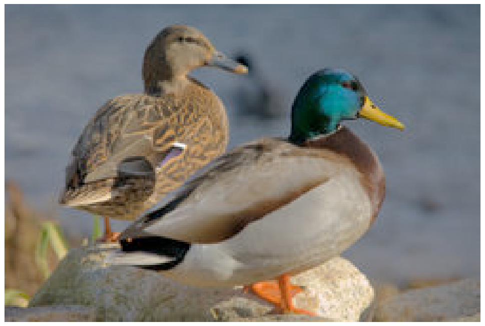
Flow

```
# Two ways: && and,  || or
if a == 1 && b == 2 and c == 3
  …
elsif s == ‘Fred’ or s == ‘Jane’
  …
else
  …
end

case text
when /"([^"\\]*(\\.[^"\\"]*)*)"/
  value = $1
  token = :string
when /^([a-zA-Z]+)$/
  value = $1
  token = :symbol
end

take_out_garbadge if age >= 7
clean_your_room unless age < 6
x = name == ‘Fred’ ? 10 : 5

# We also have while, until, unless, …
```

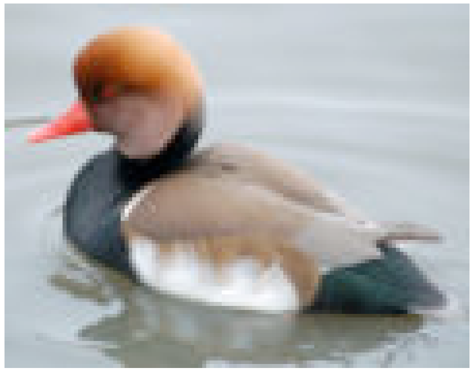
Duck Typing

If it looks like a duck, quacks like a duck, then …

```
class Mallard
   def quack; "Quack"; end
   def walk; 'waddle'; end
end
```



```
class Pochard
  def quack; "Qvack"; end
  def walk; 'waddle'; end
end
```



Duck Typing (p2)

We don’t ask if an Object is-a Duck, we ask does it behave like a duck. A goose is not a duck because it can’t quack.

```
class Goose
  def honk; “honk”; end
  def walk; ‘waddle’; end
end

birds = [Mallard.new, Pochard.new, Goose.new]

# Find the ducks
ducks = birds.select do |bird|
  bird.respond_to? :quack and bird.walk == ‘waddle’
end
ducks.each { |duck| puts duck.quack }
⇒ Quack
⇒ Qvack
```

A Real Example

Interchangeable Objects: Array and String
No is-a relationship

```
a = Array.new
1.upto(10) { |i| a << i }

a => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

s = String.new
1.upto(10) { |i| s << i }

s => "\001\002\003\004\005\006\a\b\t\n"

class Adder
  def initialize; @sum = 0; end
  def <<(v); @sum += v; end
  def inspect; @sum.to_s; end
end

t = Adder.new
1.upto(10) { |i| a << i }

a => 55
```

Modules

- Modules serve a dual purpose
  - Namespace
  - Mixin
- As a namespace:

```
module Math
  PI = 3.14159265359979
  class Point; …; end
  def sin(x); …; end
  def cos(x); …; end
  module_function :sin, :cos
end
```

Mixins

```
module TimeStamp
  def puts(string)
    super("#{Time.new.to_s}: #{string}")
  end
end


class MyClass
  def initialize
    puts "Initializing"
  end
end

MyClass.new

class MyClass
  include TimeStamp
end

MyClass.new


# Generates
Initializing
Wed Aug 08 10:41:19 -0700 2007: Initializing
```

Anonymous

Every object can have it’s own class

```
module TimeStamp
  def puts(string)
    super("#{Time.new.to_s}: #{string}")
  end
end

s = ‘Hello’
s.extend TimeStamp
s.puts s

# Generates
Wed Aug 08 10:41:19 -0700 2007: Hello

class << s
  def puts(s); super s + ‘ - xxx’; end
end

s.puts s

# Generates
Wed Aug 08 10:41:19 -0700 2007: Hello - xxx
```

Enumerations

The best way to iterate…

```
[1, 2, 3].each { |v| puts v }
=>
1
2
3

{ ‘Hello’ => 1, ‘There’ => 2}.each { |k, v| puts “#{k}: #{v}” }
=>
Hello: 1
There: 2

Array.ancestors
=> [Array, Enumerable, Object, Kernel]

Hash.ancestors
=> [Hash, Enumerable, Object, Kernel]

String.ancestors
=> [String, Enumerable, Comparable, Object, Kernel]

“a\nb\n”.map { |l| “line: #{l}” }
=> [“line: a\n”, “line: b\n”]
```

DIY Enumerations

```
class Alphabet
  def each
    ('a'..'z').each do |letter|
      yield letter
    end
  end
End

a = Alphabet.new
a.each { |l| puts l }
a
…
z

class Alphabet; include Enumerable; end
a.map { |l| "Letter #{l}" }
=> ["Letter a", "Letter b", …, "Letter z"]

a.zip([1, 2, 3])
=> [["a", 1], ["b", 2], ["c", 3], ["d", nil], …, ["z", nil]]
```

Most Mixins Require Minimal Methods
Comparible: <=>, Observable: update, …

Meta-Programming

Implementing DSLs

```
class Account < ActiveRecord::Base
  attr_accessor :access
  attr_reader :instance_state

  has_many :transactions
  belongs_to :user
end
```

DIY Meta-Programming

Simple attr reader example

```
module Reader
  def my_attr_reader(*ivars)
    ivars.each do |ivar|
       self.class_eval <<-EOT
         def #{ivar}; @#{ivar}; end
       EOT
    end
  end
end

Class Account
  extend Reader
  my_attr_reader :balance, :name
  def initialize(name); @balance = 0.0; @name = name; end
  def deposit(amt); @balance += amt; end
  def withdraw(amt); @balance -= amt; end
end
```

DIY Meta-Programming

Let’s try it out..

```
account = Account.new(‘fred’)
puts account.balance
⇒ 0.0

account.deposit(100)
puts account.balance
⇒ 100.0

account.withdraw(50)
puts account.balance
⇒ 50.0

account.withdraw(200)
puts account.balance
⇒ -150.0

puts account.name
=> fred
```

Exceptions

```
begin
  File.open(‘xxx’) do |f|
    …
  end

rescue SyntaxError
  raise

rescue IOError
  STDERR.puts “Error: #{$!}”
  retry

ensure
  @done = true
end

# Every method has an implied block
def foo
  File.open(‘xxx’) do |f|
    …
  end

rescue
  sleep 10
  retry
end

text = f.read rescue nil
```

Advanced

- There's Even More…
  - Threads
  - method_missing
  - Continuations
  - Easy C Extension
  - Lots of open-source packages (gems)
- The Future
  - Matz's Christmas Present: Ruby 2.0 (YARV)
  - JRuby (Real JIT support coming soon…)

RAD Lab
UC Berkeley

Questions