Ruby for Java Programmers

CS 169 Spring 2012

Armando Fox

Outline

- Three pillars of Ruby (§3.1)
- Everything is an object, and every operation is a method call (§3.2–3.3)
- OOP in Ruby (§3.4)
- Reflection and metaprogramming (§3.5)
- Functional idioms and iterators (§3.6)
- Duck typing and mix-ins (§3.7)
- Blocks and Yield (§3.8)
- Rails Basics: Routes & REST (§3.9)
- Databases and Migrations (§3.10)

Ruby 101

(ELLS §3.1)

Armando Fox

Ruby is...

- Interpreted
- Object-oriented
  - Everything is an object
  - Every operation is a method call on some object
- Dynamically typed: objects have types, but variables don't
- Dynamic
  - add, modify code at runtime (metaprogramming)
  - ask objects about themselves (reflection)
  - in a sense *all* programming is metaprogramming

Naming conventions

- ClassNames use UpperCamelCase

```
class FriendFinder ...  end
```

- methods & variables use snake_case

```
def learn_conventions  ...  end
def faculty_member?  ...  end
def charge_credit_card!  ...  end
```

- CONSTANTS (scoped) & $GLOBALS (not scoped)

```
TEST_MODE = true          $TEST_MODE = true
```

- *symbols:* immutable string whose value is itself

```
favorite_framework = :rails
:rails.to_s == "rails"
"rails".to_sym == :rails
:rails == "rails"  # => false
```

Variables, Arrays, Hashes

- There are no declarations!
  - local variables must be assigned before use
  - instance & class variables ==`nil` until assigned
- OK: `x = 3; x = 'foo'`
- Wrong: `Integer x=3`
- Array:
```
x = [1,'two',:three]
x[1] == 'two' ; x.length==3
```
- Hash:
```
w = {'a'=>1, :b=>[2, 3]}
w[:b][0] == 2
w.keys == ['a', :b]
```

Methods

- Everything (except fixnums) is pass-by-reference

```
def foo(x,y)
  return [x,y+1]
end

def foo(x,y=0)  # y is optional, 0 if omitted
  [x,y+1]       # last exp returned as result
end

def foo(x,y=0) ; [x,y+1] ; end
```

- Call with: `a,b = foo(x,y)`
  or `a,b = foo(x)` when optional arg used

Basic Constructs

- Statements end with ';' or newline, but can span line if parsing is unambiguous

✔
```
raise("Boom!") unless
  (ship_stable)
```
✖
```
raise("Boom!")
  unless
(ship_stable)
```

- Basic Comparisons & Booleans:

```
== != < >  =~  !~  true  false nil
```

- Th
-

if *cond* (or **unless** *cond*)	**while** *cond* (or **until** *cond*)
statements	*statements*
[**elsif** *cond*	**end**
statements]	**1.upto(10) do |i| . . . end**
[**else**	**10.times do. . . end**
statements]	*collection*.**each do |elt|. . . end**
end	

Strings & Regular Expressions

(try rubular.com for your regex needs!)

```
"string", %Q{string}, 'string', %q{string}
a=41 ; "The answer is #{a+1}"
```

- match a string against a regexp:

```
"fox@berkeley.EDU" =~ /(.*)@(.*)\.edu$/i
/(.*)@(.*)\.edu$/i =~ "fox@berkeley.EDU"
```

  - If no match, value is false
  - If match, value is non-false, and `$1...$n` *capture* parenthesized groups (`$1` == 'fox', `$2` == 'berkeley')

`/(.*)$/i` or `%r{(.*)$}i`

or `Regexp.new('(.*)$', Regexp::IGNORECASE)`

- A real example...

http://pastebin.com/hXk3JG8m

```
rx = {:fox=>/^arm/,
      'fox'=>[%r{AN(DO)$}, /an(do)/i]}
```

Which expression will evaluate to non-`nil`?

- ☐ `"armando" =~ rx{:fox}`
- ☐ `rx[:fox][1] =~ "ARMANDO"`
- ☐ `rx['fox'][1] =~ "ARMANDO"`
- ☐ `"armando" =~ rx['fox', 1]`

Everything is an object, Every operation is a method call

(ELLS §3.2-3.3)

Armando Fox

Everything is an object; (almost) everything is a method call

- Even lowly integers and nil are true objects:

```
57.methods
57.heinz_varieties
nil.respond_to?(:to_s)
```

- Rewrite each of these as calls to send:
  - Example: `my_str.length  =>  my_str.send(:length)`

```
1 + 2                          1.send(:+, 2)
my_array[4]                    my_array.send(:[], 4)
my_array[3] = "foo"            my_array.send(:[]=, 3,"foo")
if (x == 3)  ....              if (x.send(:==, 3)) ...
my_func(z)                     self.send(:my_func, z)
```

- in particular, things like “implicit conversion” on comparison is *not in the type system, but in the instance methods*

REMEMBER!

- `a.b` means: call method `b` on object `a`
  - `a` is the *receiver* to which you *send* the method call, assuming `a` will *respond to* that method

✘ *does not mean:* `b` is an instance variable of `a`

✘ *does not mean*: `a` is some kind of data structure that has `b` as a member

Understanding this distinction will save you from much grief and confusion

Example: every operation is a method call

```
y = [1,2]
y = y + ["foo",:bar] # => [1,2,"foo",:bar]
y << 5               # => [1,2,"foo",:bar,5]
y << [6,7]           # => [1,2,"foo",:bar,5,[6,7]]
```

- "<<" *destructively modifies* its receiver, "+" does not
  - destructive methods often have names ending in "!"
- Remember! These are nearly all *instance methods* of Array —*not* language operators!
- So `5+3`, `"a"+"b"`, and `[a,b]+[b,c]` are all *different* methods named '+'
  - `Numeric#+`, `String#+`, and `Array#+`, to be specific

Hashes & Poetry Mode

```
h = {"stupid" => 1, :example=> "foo" }
h.has_key?("stupid") # => true
h["not a key"]       # => nil
h.delete(:example) # => "foo"
```

- Ruby idiom: “poetry mode”
  - using hashes to pass “keyword-like” arguments
  - omit hash braces when **last** argument to function is hash
  - omitting parens around function arguments

```
link_to("Edit”,{:controller=>'students', :action=>'edit'})
link_to "Edit", :controller=>'students', :action=>'edit'
```

- When in doubt, parenthesize defensively

Poetry mode in action

```
a.should(be.send(:>=,7))
a.should(be() >= 7)
a.should be >= 7
```

```
(redirect_to(login_page)) and return()
  unless logged_in?
redirect_to login_page and return
  unless logged_in?
```

```
def foo(arg,hash1,hash2)
  ...
end
```

Which is *not* a legal call to `foo()`:

- ☐ `foo a, {:x=>1,:y=>2}, :z=>3`
- ☐ `foo(a, :x=>1, :y=>2, :z=>3)`
- ☐ `foo(a, {:x=>1,:y=>2},{:z=>3})`
- ☐ `foo(a, {:x=>1}, {:y=>2,:z=>3})`

Ruby OOP
(ELLS §3.4)

Armando Fox

Classes & inheritance

```ruby
class SavingsAccount < Account    # inheritance
  # constructor used when SavingsAccount.new(...) called
  def initialize(starting_balance=0) # optional argument
    @balance = starting_balance
  end
  def balance # instance method
    @balance   # instance var: visible only to this object
  end
  def balance=(new_amount)  # note method name: like setter
    @balance = new_amount
  end
  def deposit(amount)
    @balance += amount
  end
  @@bank_name = "MyBank.com"    # class (static) variable
  # A class method
  def self.bank_name   # note difference in method def
    @@bank_name
  end
  # or: def SavingsAccount.bank_name ; @@bank_name ; end
end
```

http://pastebin.com/m2d3myyP

Which ones are correct:

(a) `my_account.@balance`
(b) `my_account.balance`
(c) `my_account.balance()`

- ☐ All three
- ☐ Only (b)
- ☐ (a) and (b)
- ☐ (b) and (c)

Instance variables: shortcut

```
class SavingsAccount < Account
  def initialize(starting_balance)
    @balance = starting_balance
  end
  def balance
    @balance
  end
  def balance=(new_amount)
    @balance = new_amount
  end
end
```

Instance variables: shortcut

```
class SavingsAccount < Account
  def initialize(starting_balance)
    @balance = starting_balance
  end

  attr_accessor :balance


end
```

`attr_accessor` is just a *plain old method that uses metaprogramming*...**not** part of the language!

```
class String
  def curvy?
   !("AEFHIKLMNTVWXYZ".include?(self.upcase))
  end
end
```

- `String.curvy?("foo")`
- `"foo".curvy?`
- `self.curvy?("foo")`
- `curvy?("foo")`

Review: Ruby's Distinguishing Features (So Far)

- Object-oriented with **no** multiple-inheritance
  - *everything* is an object, even simple things like integers
  - class,instance variables *invisible* outside class
- Everything is a method call
  - usually, only care if *receiver responds to method*
  - most "operators" (like +, ==) actually instance methods
  - Dynamically typed: objects have types; variables don't
- Destructive methods
  - Most methods are nondestructive, returning a new copy
  - Exceptions: <<, some destructive methods (eg `merge` vs. `merge!` for hash)
- Idiomatically, {} and () sometimes optional

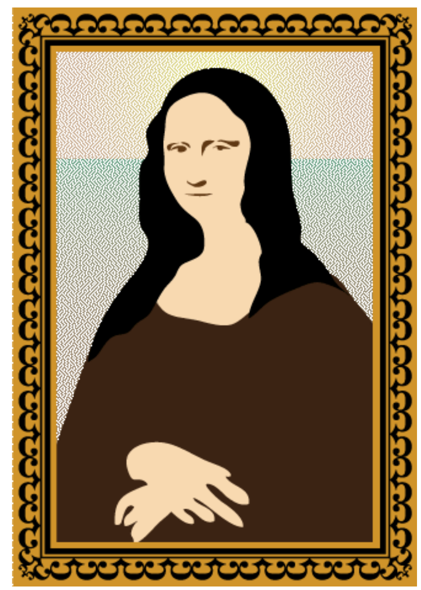
All Programming is Metaprogramming

(ELLS §3.5)

Armando Fox

An international bank account!

```
acct.depo    00)              eposit $100
acct.dep    t(    os_to_d    lars(20))
    # ab    $25
```



An international bank account!

```
acct.deposit(100)     # deposit $100
acct.deposit(20.euros)    # about $25
```

- No problem with open classes....

```
class Numeric
  def euros ; self * 1.292 ; end
end
```

http://pastebin.com/f6WuV2rC

- But what about

```
acct.deposit(1.euro)
```

http://pastebin.com/WZGBhXci

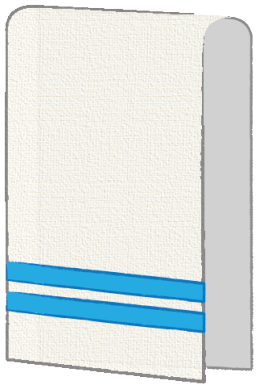
The power of `method_missing`

- But suppose we also want to support

```
acct.deposit(1000.yen)
acct.deposit(3000.rupees)
```

- Surely there is a DRY way to do this?

http://pastebin.com/agjb5qBF



Introspection & Metaprogramming

- You can ask Ruby objects questions about themselves at runtime
- You can use this information to *generate new code* (methods, objects, classes) at runtime
- You can "reopen" any class at any time and add stuff to it.
  - this is *in addition* to extending/subclassing

Suppose we want to handle

```
5.euros.in(:rupees)
```

What mechanism would be most appropriate?

- ☐ Change `Numeric.method_missing` to detect calls to 'in' with appropriate args
- ☐ Change `Numeric#method_missing` to detect calls to 'in' with appropriate args
- ☐ Define the method `Numeric#in`

- ☐ Define the method `Numeric.in`

Blocks, Iterators, Functional Idioms

(ELLS §3.6)

Armando Fox

Loops—but don't think of them that way

```
["apple", "banana", "cherry"].each do |string|
  puts string
end

for i in (1..10) do
  puts i
end

1.upto 10 do |num|
  puts num
end

3.times {  print "Rah, " }
```

If you're iterating with an index, you're probably doing it wrong

- *Iterators* let objects manage their own traversal
- `(1..10).each do |x| ... end`
  `(1..10).each { |x| ... }`
  `1.upto(10) do |x| ... end`
  => range traversal
- `my_array.each do |elt| ... end`
  => array traversal
- `hsh.each_key do |key| ... end`
  `hsh.each_pair do |key,val| ... end`
  => hash traversal
- `10.times {...} # => iterator of arity zero`
- `10.times do ... end`

“Expression orientation”

```
x = ['apple','cherry','apple','banana']
x.sort # => ['apple','apple','banana','cherry']
x.uniq.reverse # => ['banana','cherry','apple']
x.reverse!  # => modifies x
x.map do |fruit|
  fruit.reverse
end.sort
  # => ['ananab','elppa','elppa','yrrehc']
x.collect { |f| f.include?("e") }
x.any? { |f| f.length > 5 }
```

- A real life example....

http://pastebin.com/Aqgs4mhE

Which string will *not* appear in the result of:

```
['banana','anana','naan'].map do |food|
  food.reverse
end.select { |f| f.match /^a/ }
```

- ☐ naan
- ☐ ananab
- ☐ anana
- ☐ The above code won't run due to syntax error(s)

Mixins and Duck Typing

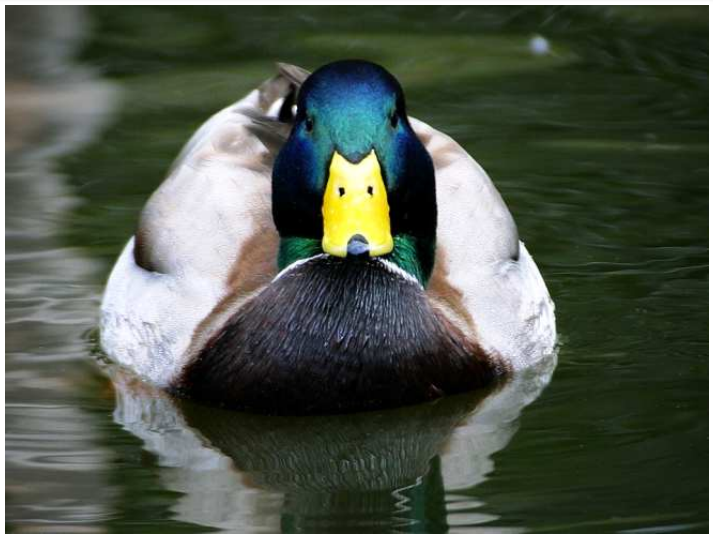
(ELLS §3.7)

Armando Fox

What is “duck typing”?

- If it responds to the same methods as a duck...it might as well be a duck
- More than just overloading; similar to Java Interfaces
- Example: `my_list.sort`

```
[5, 4, 3].sort
["dog", "cat", "rat"].sort
[:a, :b, :c].sort
IO.readlines("my_file")
```



Modules

- A *module* is a collection of class & instance methods that are not actually a class
  - you can't instantiate it
  - Some modules are *namespaces*, similar to Python: `Math::sin(Math::PI / 2.0)`
- The more interesting ones let you *mix the methods into* a class:

```
class A < B ; include MyModule ; end
```

  - `A.foo` will search `A`, then `MyModule`, then `B`
  - `sort` is actually defined in module `Enumerable`, which is *mixed into* Array by default

A Mix-in Is A Contract

- Example: `Enumerable` assumes objects of target class respond to `each`
  - ...provides `all?, any?, collect, find, include?, inject, map, partition`, ...
- Example: `Comparable` assumes that objects of target class respond to `<=>`
  - provides `< <= => > == between?` for free
- `Enumerable` also provides `sort`, which requires *elements* of target class (things returned by `each`) to respond to `<=>`

Class of objects doesn't matter: only methods to which they respond

Example: sorting a file

- Sorting a file
  - `File.open` returns an `IO` object
  - `IO` objects respond to `each` by returning each line as a `String`
- So we can say `File.open ('filename.txt').sort`
  - relies on `IO#each` and `String#<=>`
- Which lines of file begin with vowel?

```
File.open('file').
  select { |s| s =~ /^[aeiou]/i }
```

```
a = SavingsAccount.new(100)
b = SavingsAccount.new(50)
c = SavingsAccount.new(75)
```
What's result of `[a,b,c].sort`

- ☐ Works, because account balances (numbers) get compared
- ☐ Doesn't work, but would work if we passed a comparison method to `sort`
- ☐ Doesn't work, but would work if we defined `<=>` on `SavingsAccount`
- ☐ Doesn't work: `SavingsAccount` isn't a basic Ruby type so can't compare them

Making accounts comparable

- Just define `<=>` and then use the `Comparable` module to get the other methods
- Now, an `Account` quacks like a numeric ☺

http://pastebin.com/itkpaqMh

When Module? When Class?

- Modules reuse *behaviors*
  - high-level behaviors that could conceptually apply to many classes
  - Example: `Enumerable, Comparable`
  - Mechanism: mixin `(include Enumerable)`
- Classes reuse *implementation*
  - subclass reuses/overrides superclass methods
  - Mechanism: inheritance `(class A < B)`
- Remarkably often, we will *prefer composition over inheritance*

yield()
(ELLS §3.8)

Armando Fox

Blocks (anonymous λ)

```
(map '(lambda (x) (+ x 2)) mylist )
mylist.map { |x| x+2 }

(filter '(lambda (x) (even? x)) mylist)
mylist.select do |x| ; x.even? ; end

(map
  '(lambda (x) (+ x 2))
  (filter '(lambda (x) (even? x)) mylist))
mylist.select {|x| x.even?}.map {|x| x+2 }
```

Turning iterators inside-out

- Java:
  - You hand me each element of that collection in turn.
  - I'll do some stuff.
  - Then I'll ask you if there's any more left.
- Ruby:
  - Here is some code to apply to every element of the collection.
  - *You* manage the iteration or data structure traversal.
- Let's do an example...

http://pastebin.com/T3JhV7Bk

Iterators are just one nifty use of *yield*

```
# in some other library
def before_stuff
  ...before code...
end
def after_stuff
  ...after code...
end

# in your code
def do_everything
  before_stuff()
  my_custom_stuff()
  after_stuff()
end
```

Without yield(): expose 2 calls in other library

```
# in some other library
def around_stuff
  ...before code...
  yield
  ...after code...
end

# in your code
def do_everything
  around_stuff do
    my_custom_stuff()
  end
end
```

With yield(): expose 1 call in other library

Blocks are Closures

- A *closure* is the set of all variable bindings you can "see" at a given point in time
  - In Scheme, it's called an *environment*
- *Blocks are closures:* they carry their environment around with them

  http://pastebin.com/zQPh70NJ

- Result: blocks can help reuse by separating *what to do* from *where & when to do it*
  - We'll see various examples in Rails

In Ruby, every _____ accepts a(n) _____, but not vice-versa.

- ☐ yield() statement; iterator
- ☐ closure; iterator
- ☐ block; iterator
- ☐ iterator; block

Summary

- Duck typing for re-use of behaviors
  - In Ruby, it's achieved with "mix-ins" via the Modules mechanism, and by the "everything is a message" language semantics
- Blocks and iterators
  - Blocks are anonymous lambdas that *carry their environment around with them*
  - Allow "sending code to where an object is" rather than passing an object to the code
  - Iterators are an important special use case