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'
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

```
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)

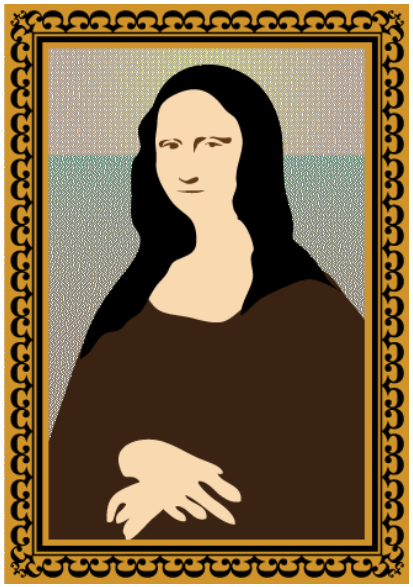
- Object-oriented with **no** multiple-inheritance
- *everything* is an object, even simple things like integers
- class,instance variables *invisible* outside class
- Everything is a method call
- usually, only care if *receiver responds to method*
- most "operators" (like +, ==) actually instance methods
- 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.deposit(100)     # deposit $100
acct.deposit(euros_to_dollars(20))
    # about $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

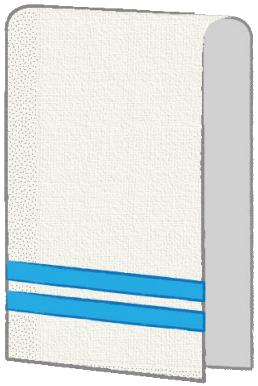
- But suppose we also want to support

acct.deposit(1000.yen)

acct.deposit(3000.rupees)

- Surely there is a DRY way to do this?

http://pastebin.com/agjb5qBF



http://pastebin.com/HJTvUid5

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 handle5.euros.in(:rupees)

What change to Numeric 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)