All Programming is Metaprogramming

(Engineering Software as a Service §3.5)

Armando Fox

Metaprogramming & Reflection

- *Reflection* lets us ask an object questions about itself and have it modify itself
- *Metaprogramming* lets us define new code at runtime

- How can these make our code DRYer, more concise, or easier to read?
  - (or are they just twenty-dollar words to make me look smart?)

An international bank account

```
acct.deposi      00)                  deposit $100
acct.depos       (      os_to_      llars(20))
acct.depo        t(Cu      ncyC      verter.new(
   :euros,       ))
```

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

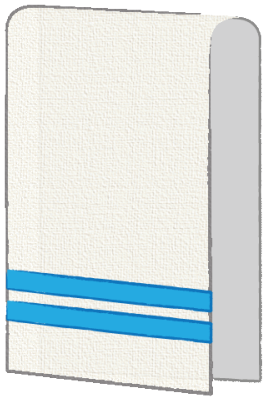
The power of `method_missing`

- 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

Reflection & Metaprogramming

- You can ask Ruby objects questions about themselves at runtime *(introspection)*
- You can use this information to *generate new code* (methods, objects, classes) at runtime (*reflection)*
- …so can have *code that writes code* (*metaprogramming)*
- You can "reopen" any class at any time and add stuff to it.
  - …*in addition* to extending/subclassing it!

Suppose we want to handle

```
5.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

(Engineering Software as a Service §3.6)

Armando Fox

Functionally flavored

- *How* can techniques from *functional programming* help us rethink basic programming concepts like iteration?

- And *why* is it worth doing that?

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

(Engineering Software as a Service §3.7)

Armando Fox

So what if you're not my type

- Ruby emphasizes

  "What methods do you respond to?"

  over

  "What class do you belong to?"

- *How does this encourage productivity through reuse?*

What is “duck typing”?

- If it responds to the same methods as a duck...it might as well be a duck
- Similar to Java Interfaces but easier to use
- Example: `my_list.sort`

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

Modules

- Collection of methods that aren't a class
  - you can't instantiate it
  - Some modules are *namespaces*, similar to Python: `Math::sin(Math::PI / 2.0)`
- Important use of modules: *mix its methods into* a class:

```
class A ; include MyModule ; end
```

  - `A.foo` will search `A`, then `MyModule`, then `method_missing` in `A` & `B`, then `A`'s ancestor
  - `sort` is actually defined in module `Enumerable`,

which is *mixed into* Array by default

A Mix-in is a Contract

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

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()
(Engineering Software as a Service §3.8)

Armando Fox

Inelegant, this

```
ArrayList aList;
Iterator it = aList.iterator();
while (it.hasNext()) {
  Object element = it.getNext();
  // do some stuff with element
}
```

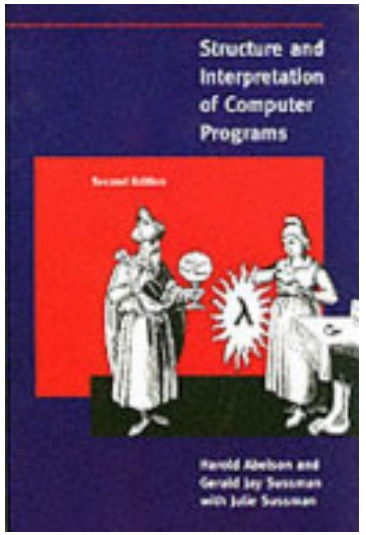
- Goal of the code: do stuff with elements of `aList`
- But iterator logic is all jumbled up with the code

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. Give me each element to do stuff to.
- Let's do an example...

http://pastebin.com/T3JhV7Bk

Iterators are just one nifty use of *yield*

```
# in File class
def open(filename)
  ...open a file...
end
def close
  ...close a file...
end

# in your code
def do_everything
  f = File.open("foo")
  my_custom_stuff(f)
  f.close()
end
```

Without yield(): expose 2 calls in other library

```
# in some other library
def open(filename)
  ...before code...
  yield file_descriptor
  ...after code...
end


# in your code
def do_everything
  File.open("foo") do |f|
    my_custom_stuff(f)
  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 encourages behavior reuse
  - "mix-in" a module and rely on "everything is a method call—do you respond to this method?"
- 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

Summary (cont.)