LABORATORIO APPLICAZIONE SW E SICUREZZA INFORMATICA

ROBERTO BERALDI

Variables	local_variable, @@class_variable, @instance_variable
Constants	ClassName, CONSTANT, $GLOBAL, $global
Booleans	false, nil are false; true and *everything else* (zero, empty string, etc.) is true.
Strings and Symbols	"string", 'also a string', %q{like single quotes}, %Q{like double quotes}, :symbol special characters (\n) expanded in double-quoted but not single-quoted strings
Expressions in *double-quoted* strings	@foo = 3 ; "Answer is #{@foo}"; %Q{Answer is #{@foo+1}}
Regular expression matching	"hello" =~ /lo/ or "hello".match(Regexp.new 'lo')
Arrays	a = [1, :two, 'three'] ; a[1] == :two
Hashes	h = {:a =>1, 'b' =>"two"} ; h['b'] == "two" ; h.has_key?(:a) == true
Hashes (alternate notation, Ruby 1.9+)	h = {a: 1, 'b': "two"}
Instance method	def method(*arg, arg*)...end *(use* ***args** *for variable number of arguments*
Class (static) method	def ClassName.method(*arg, arg*)...end, def self.method(*arg, arg*)...end
Special method names *Ending these methods' names in* ? *and* ! *is optional but idiomatic*	def setter=(*arg, arg*)...end def boolean_method?(*arg, arg*)...end def dangerous_method!(*arg, arg*)...end

Conditionals	**Iteration** (see Section 3.6)	**Exceptions**
if *cond* (or **unless** *cond*) *statements* [**elsif** *cond* *statements*] [**else** *statements*] **end**	**while** *cond* (or **until** *cond*) *statements* **end** **1.upto(10) do** |i|...**end** **10.times do**...**end** *collection*.**each do** |elt|...**end**	**begin** *statements* **rescue AnError => e** **e** *is an exception of class* **AnError**; *multiple* **rescue** *clauses OK* [**ensure** *this code is always executed*] **end**

RUBY (ALCUNE CARATTERISTICHE)

- DRY (Don't Repeat Yourself)

- Interpretato (IRB)
- Orientato agli oggetti (Object Oriented): Ogni entità è un oggetto
- Ogni operazione è una chiamata al metodo su un oggett
- Dinamico: ossia è possible aggiungere e/o modificare codice in fase di esecuzione (metaprogrammazione)
- …Ispezionare gli oggetti mentre il programma gira (reflection)

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:* like 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

```
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
- `a,b = foo(x,y)`
  `a,b = foo(x)` when optional arg used

BASIC CONSTRUCTS

- Basic Comparisons & Booleans:
  `==  !=  <  >  =~  !~  true  false nil`
- The usual control flow constructs

if *cond* (or **unless** *cond*)
  statements
[**elsif** *cond*
  statements]
[**else**
  statements]
end

while *cond* (or **until** *cond*)
  statements
end
1.upto(10) do |i| . . . end
10.times do. . . end
collection.**each do |elt|. . . end**

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
my_array[4]
my_array[3] = "foo"
if (x == 3)  ....
my_func(z)
```

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

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]]
```

- Remember! These are *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'
```

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?
```

RUBY OOP

(ENGINEERING SOFTWARE AS A SERVICE § 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
```

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` *uses metaprogramming..*

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.deposit(100)      # deposit
acct.deposit(euros_to_dollars(20))
acct.deposit(CurrencyConverter.new(
  :euros, 20))
```

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)
```

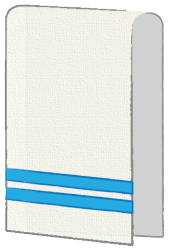
http://pastebin.com/WZGBhXci

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!

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
  my_array.each_with_index do |elt, index|
    ...
  end
  ```
  => array traversal
- ```
  hsh.each_key do |key| ... end
  hsh.each_pair do |key, val| ... end
  ```
  => hash traversal
- ```
  10.times {...} # => iterator of arity zero
  ```

“EXPRESSION ORIENTATION”

```ruby
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

MIXINS AND DUCK TYPING

(ENGINEERING SOFTWARE AS A SERVICE § 3.7)

ARMANDO FOX

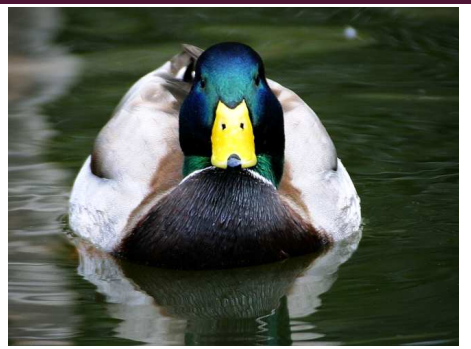
SO WHAT IF YOU'RE NOT MY TYPE

- 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 }
```

MAKING ACCOUNTS COMPARABLE

- Just define `<=>` and then use the `Comparable` module to get the http://pastebin.com/itkpaqMh
- Now, an `Account` quacks like a numeric ☺

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*

MIXINS/INHERITANCE GONE WRONG

- https://www.destroyallsoftware.com/blog/2011/one-base-class-to-rule-them-all
- `Base.new.methods.count => 6947`

- `class Cantaloupe < Base ; end`
- `Cantaloupe.new.size => 0 # ???`

- Which `#size`? It mixed in `Hash#size, Array#size, Enumerable#size, String#size`
- Be judicious with mixins!

REMEMBER PYTHON ITERATORS?

```
class Fib:
    def
__init__(how_many):
    def __iter__(self):
    def __next__(self):

for n in Fib(5):
    # do stuff with n
```

```
class Fib
  def initialize(how_many)
  …
  end
  def each ; … ; end
end
Fib.new(5).each do |n|
  # do stuff with n
end
```

- Goal: do stuff with elements of a collection
- Ruby's version:
  - *the collection itself provides iteration* by defining `each`
  - You pass *block* (anonymous lambda) to say what to do with each element

http://pastebin.com/T3JhV7Bk

BUT, ITERATORS ARE JUST *ONE* NIFTY USE OF `YIELD`

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

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

Yields 1 argument to its block

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

GENERATE A FANCY TABLE

```
# in CoolTable class
def initialize()
  ...output table-start ma
end
def end_table
  ...output table-end mar
end

# in your code
def make_cool_table
  t = Table.new()
  t.build_table_body(...)
  t.end_table()
end
```

Without yield(): expose 2 calls in other library

Yields no arguments

```
# in CoolTable class
def initialize(args)
  ...output table-start markup...
  yield
  ...output table-end markup...
end

# in your code
def make_cool_table
  t = Table.new(args) do
    build_table_body(…)
  end
end
```

With yield()

BLOCKS ARE CLOSURES

- A *closure* is the set of all variable bindings you can "see" at a given point in time
  - it "closes over" all the variables you can see, including nonlocal variables
  - Also available in Python

http://pastebin.com/zQPh70NJ

- *Ruby blocks are closures:* they carry their environment around with them
- Result: blocks can help reuse by separating *what to do* from *where & when to do it*
  - We'll see various examples in Rails

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

PEER LEARNING: RUBY

```
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]`

```
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}`

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)

CLASSES & INHERITANCE

```ruby
class SavingsAccount < Account
  # constructor used when SavingsAccount.new(...) called
  def initialize(starting_balance=0)
    @balance = starting_balance
  end
  def balance
    @balance
  end
  def balance=(new_amount)
    @balance = new_amount
  end
  def deposit(amount)
    @balance += amount
  end
  @@bank_name = "MyBank.com"
  # A class method
  def self.bank_name
    @@bank_name
  end
  # or: def SavingsAccount.bank_name ; @@bank_name ; end
end
```

Which ones are correct:

(a) `my_account.@balance`

(b) `my_account.balance`

(c) `my_account.balance()`

http://pastebin.com/m2d3myyP

An attribute on an object of class `Foo`
can be *written to* by a method in class `Bar` if:

- ☐ the attribute has a setter method defined using `attr_accessor` or `attr_writer`
- ☐ The attribute has a setter method defined explicitly, even if the method's name doesn't match the attribute's name
- ☐ The attribute is labeled `public`

- ☐ The `Foo` and `Bar` classes inherit from a common ancestor class

```
class Student
  def name
    capitalize_words(@student_name)
  end
end
```

- Illegal: accessor method must have same name as instance variable
- Illegal: accessor method must return "raw" instance variable
- Legal: can use obj.student_name to get "raw" value of this attribute
- Legal, but no way to get "raw" value of this attribute

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

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

Ruby libraries are called _____ and you manage the ones your app needs with ____

- ☐ modules; Rails
- ☐ gems; Bundler
- ☐ gems; *gemcutter*
- ☐ packages; *rpack*


END