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..*

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.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)
```

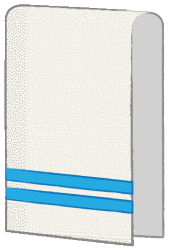
http://pastebin.com/WZGBhXci

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!