Ruby

Marek Paška

Why “scripting” language?

Language	*Statements ratio*	*Lines ratio*
C	1.00	1.00
C++	2.50	1.00
Fortran	2.50	0.80
Java	2.50	1.50
Perl	6.00	6.00
Smalltalk	6.00	6.25
Python	6.00	6.50

Ruby and Python History

- Appeared in 1995
- Designed by Yukihiro Matsumoto
- De-facto standard: C-based interpreter implementation

- Appeared in 1991
- Designed by Guido van Rossum
- De-facto standard: C-based interpreter implementation

"benevolent dictator for life"

Ruby Properties

- Very high level language
- Everything is accessible at run-time (actually no compile/run-time difference)
- Everything is object
- Built-in arraylist, hashtables

Ruby Coordinates

- Ruby is two parts Perl, one part Python, and one part Smalltalk
- But:
  - not as ugly as Perl
  - fully object-oriented (unlike Python)
  - straightforward syntax (unlike Smalltalk)

Type System

- Duck typing is a style of dynamic typing in which an object's current set of methods and properties determines the valid semantics, rather than its inheritance from a particular class.
- *If it walks like a duck and quacks like a duck, I would call it a duck.*

Type System - Classes

- class definition is never closed
- example: adding method to built-in String class

```
class String
  def twice()
    return (self + " ") * 2
  end
end

s = "hallo"
puts s.twice #prints "hallo hallo"
```

Type System - Methods

- adding method to one particular instance

```
class << s
   def twice()
      (self + "\n") * 2
   end
end

puts s.twice
```

- method alias (one page AOP)

```
class String
   alias :toString :to_s
end
```

Type System - Inheritance

- No multiple inheritance
- Modules – interfaces on steroids
- Mixins

```
module M
   def m()
      "hallo from module"
   end
end

class C
   include M
end
```

```
c=C.new
puts c.m
```

Type System - Attributes

- No verbose getters and setters

```
class Tuple
  def initialize(a,b)
    @a = a
    @b = b
  end

  attr_reader :a
  attr_accessor :b
end

f = Tuple.new(1,2)
puts f.a
f.b = 3
```

Closures

- piece of code sent as parameter

```
x = [1, 2, 3, 4]

x.each {|i| puts i} #prints all items

x2 = x.map {|i| i*i}

x3 = x.select {|i| i > 2}

puts "x3:", x3
```

Closures - transactions

```
File.open('file.txt', 'w') do |file|
  file.puts 'Wrote some text.'
end #file is automatically closed here
```

Sweet Details

- method name conventions
  - if ends with “!” then changes object state
  - if ends with “?” then returns boolean

```
s = “hallo”
s.capitalize    #returns “Hallo”, s is “hallo”
s.capitalize!   #returns “Hallo”, s is “Hallo”
s.empty?        #returns false
```

Threading

- uses user-level “green” threads
  - cheap
  - no speedup, no slowdown
  - web development: processes instead of threads
- JRuby uses Java threads
  - breaks some libraries

Strings

- No built-in unicode (because of Japan origin)
- Strings are “binary”
- Usually utf-8 encoding (like gnome)
- No “char” data type
- Unicode “broken” in many languages (Java, C#, Python)

Ruby on Rails

- Just a MVC framework
- “Convention over configuration”
  - application layout is predefined (comfortable for developers, cheap for maintenance)
- *Zero turn-around time*
- O/R mapping: design pattern “ActiveRecord”

RoR – Sequence of Operations

1. Create database schema in relation db like MySQL
2. ActiveRecord classes are generated at runtime
3. Generate CRUD version of application – scaffolding
4. Use advantage of zero turn-around time