The Ruby Language

Chauk-Mean PROUM

March 2007

Object-Oriented and Meta-Programming

Copyrights

Why this presentation ?

The latest trends in Programming Languages are :

- Dynamic Typing
- Functional Programming
- Domain Specific Language (DSL)

And

Whenever you learn a new language, it changes the way you think.

(Bruce Tate, author of "Better, Faster, Lighter Java", "Beyond Java",…, "From Java to Ruby", "Ruby on Rails : Up and Running" ,..)

History

Ruby was created in 1995 (V1.0) in Japan by Yukihiro "Matz" Matsumoto.

It has come to Western Countries only in 2000.

まつもと ゆきひろ

Ruby = Smalltalk - unfamiliar syntax
+ PERL 's scripting power
+ Python 's exceptions etc.
+ CLU 's iterator
+ a lot more good things

Freedom and Comfort

- Freedom : there is more than one way of doing things
- Comfort : the "better" way is made comfortable (a.k.a. the Ruby way)

The Result

Ruby is ...

- a Scripting Language
- a Dynamic Typing Language
- an Object Oriented Programming Language
- a good taste of Functional Programming
- a highly Reflective Language
- a base for creating Domain Specific Languages

Contents

- Ruby Description

  Scripting, Dynamic Typing, Object Oriented, Functional Programming, Reflection, DSL

- More Ruby vs. Others

  C++, Java, Python, Groovy, and PHP

- More on Ruby

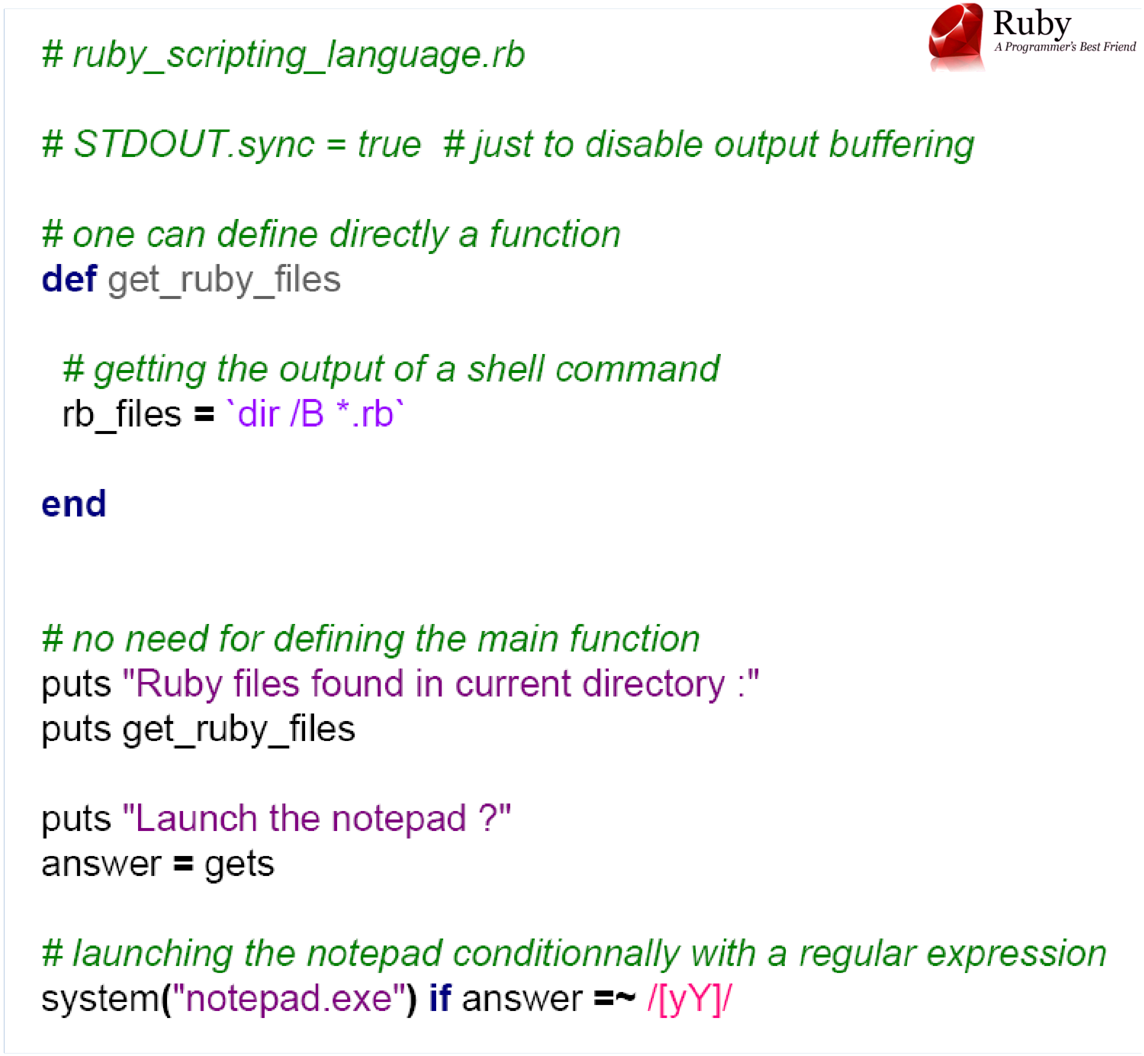
Ruby : a Scripting Language

Ruby : a Scripting Language (1/2)

A Scripting Language is for gluing existing applications/components :

- easy to write
- typically interpreted (no explicit compilation required)
- typically dynamically typed for favouring rapid development over efficiency of execution
- strong at communicating with program components written in other languages

Ruby : a Scripting Language (2/2)



```
# ruby_scripting_language.rb

# STDOUT.sync = true  # just to disable output buffering

# one can define directly a function
def get_ruby_files

  # getting the output of a shell command
  rb_files = `dir /B *.rb`

end


# no need for defining the main function
puts "Ruby files found in current directory :"
puts get_ruby_files

puts "Launch the notepad ?"
answer = gets

# launching the notepad conditionnally with a regular expression
system("notepad.exe") if answer =~ /[yY]/
```

From Perl, Ruby picks up a lot of Unix shell programming features and built-in regular expressions*.

*The power of Perl and Unix tools like sed and awk comes from their built-in support for regular expressions.

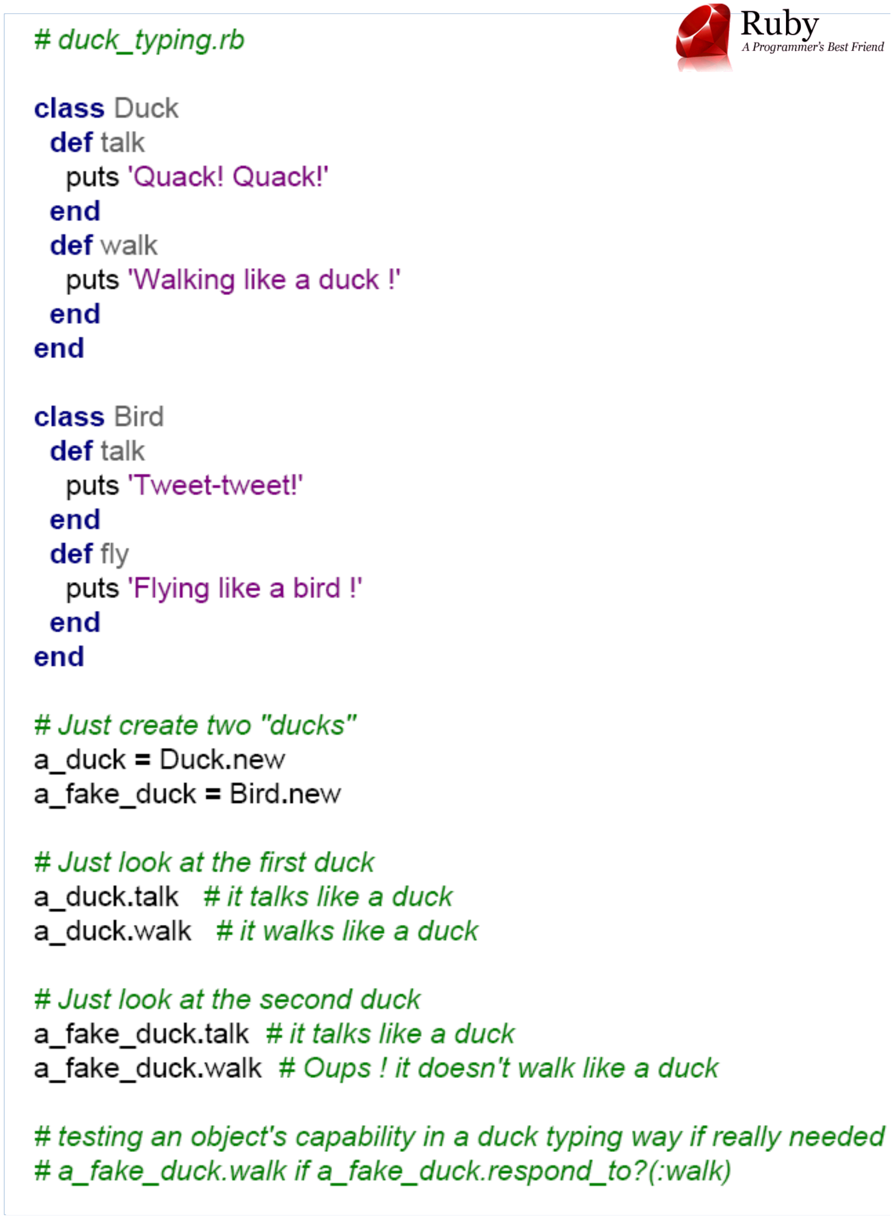
Ruby : a Dynamic Typing Language

Ruby : a Dynamic Typing Language (1/8)

Duck Typing

(Dave Thomas, author of "Programming Ruby", and "Agile Web Development with Rails")

- "If an object walks like a duck and talks like a duck, it must be a duck."
- The <u>type of an object</u> is defined by <u>what that object can do</u> (and not by its class/interface).

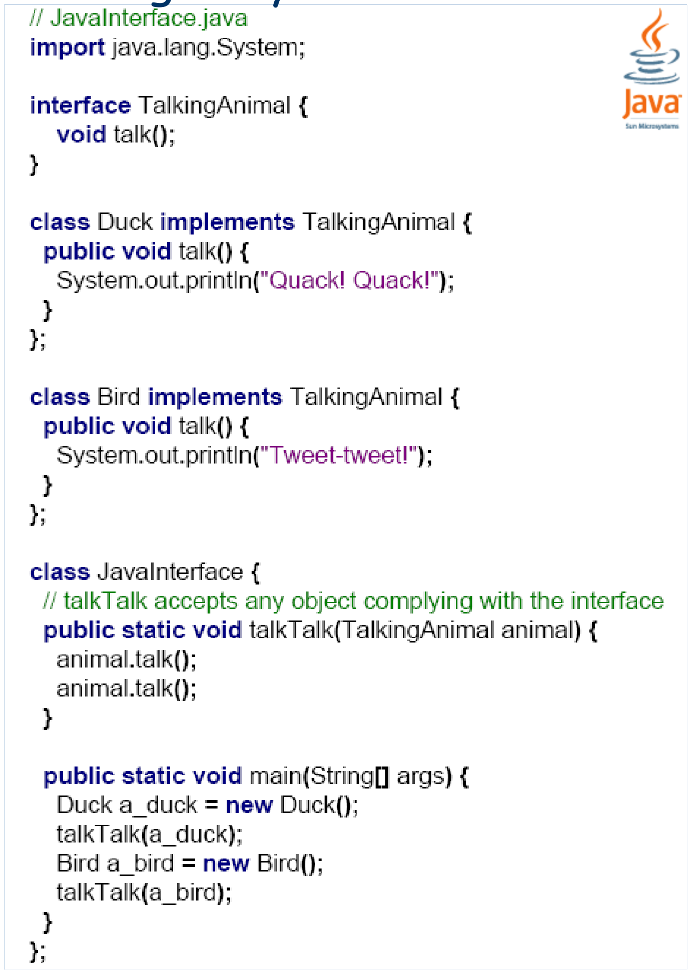


Ruby
A Programmer's Best Friend

```ruby
# duck_typing.rb

class Duck
  def talk
    puts 'Quack! Quack!'
  end
  def walk
    puts 'Walking like a duck !'
  end
end

class Bird
  def talk
    puts 'Tweet-tweet!'
  end
  def fly
    puts 'Flying like a bird !'
  end
end

# Just create two "ducks"
a_duck = Duck.new
a_fake_duck = Bird.new

# Just look at the first duck
a_duck.talk   # it talks like a duck
a_duck.walk   # it walks like a duck

# Just look at the second duck
a_fake_duck.talk  # it talks like a duck
a_fake_duck.walk  # Oups ! it doesn't walk like a duck

# testing an object's capability in a duck typing way if really needed
# a_fake_duck.walk if a_fake_duck.respond_to?(:walk)
```

Ruby : a Dynamic Typing Language (2/8)

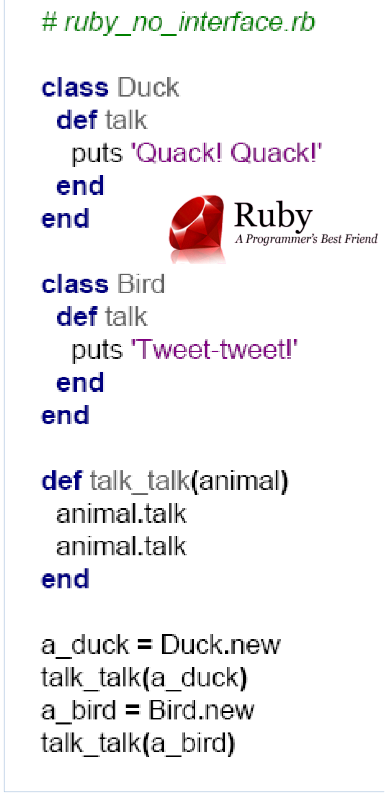
In Java, an interface allows different classes (implementations) to be used interchangeably.



```java
// JavaInterface.java
import java.lang.System;

interface TalkingAnimal {
    void talk();
}

class Duck implements TalkingAnimal {
  public void talk() {
    System.out.println("Quack! Quack!");
  }
};

class Bird implements TalkingAnimal {
  public void talk() {
    System.out.println("Tweet-tweet!");
  }
};

class JavaInterface {
  // talkTalk accepts any object complying with the interface
  public static void talkTalk(TalkingAnimal animal) {
    animal.talk();
    animal.talk();
  }

  public static void main(String[] args) {
    Duck a_duck = new Duck();
    talkTalk(a_duck);
    Bird a_bird = new Bird();
    talkTalk(a_bird);
  }
};
```

In Ruby, there is no need for interface.

Any object responding to the relevant methods is suitable.



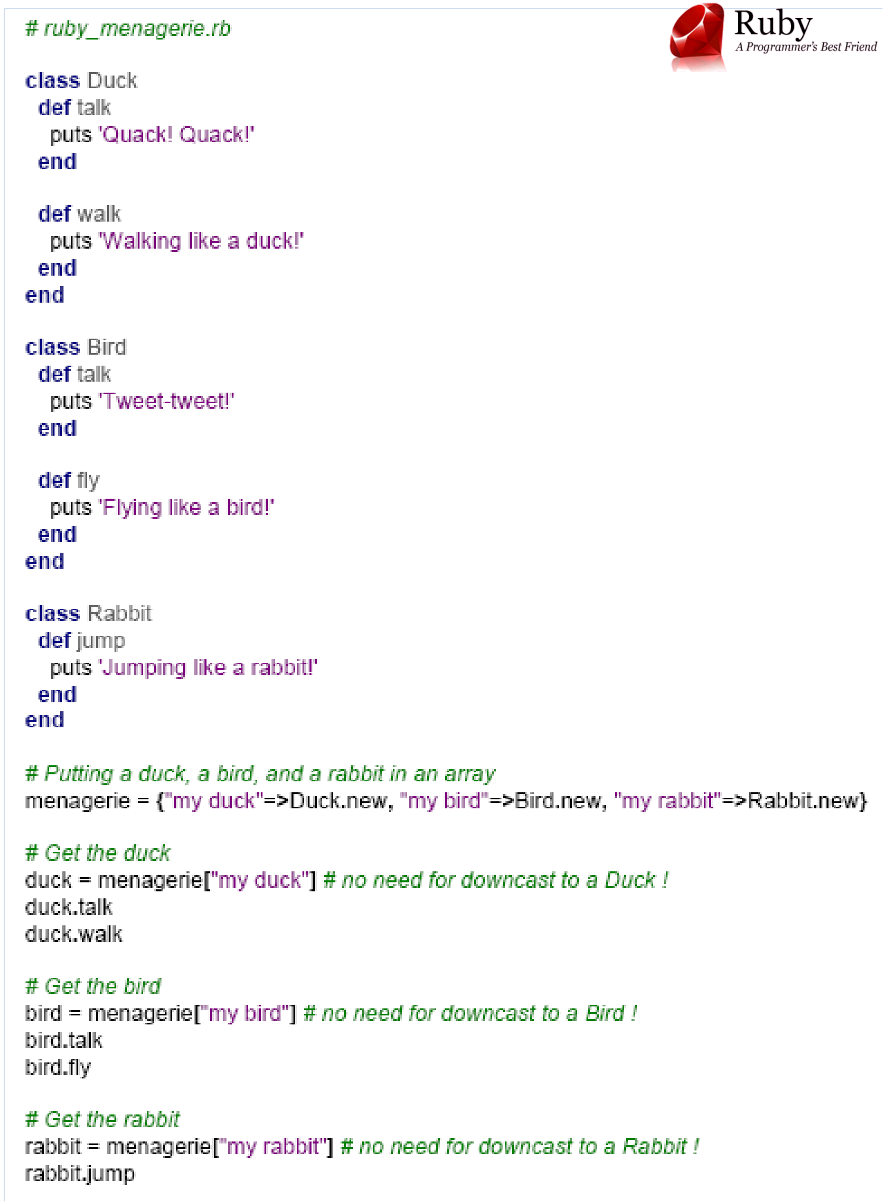
```ruby
# ruby_no_interface.rb

class Duck
  def talk
    puts 'Quack! Quack!'
  end
end

class Bird
  def talk
    puts 'Tweet-tweet!'
  end
end

def talk_talk(animal)
  animal.talk
  animal.talk
end

a_duck = Duck.new
talk_talk(a_duck)
a_bird = Bird.new
talk_talk(a_bird)
```

Ruby : a Dynamic Typing Language (3/8)

Benefits : Simplicity and Flexibility

Ruby Collections are more simple and more flexible to use than their Java counterparts :

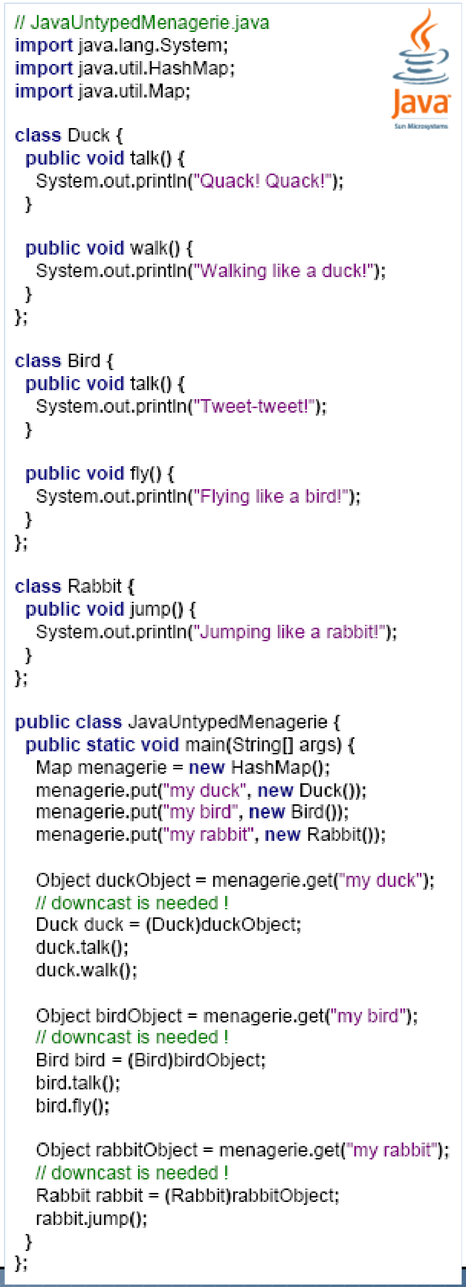
- no need for downcast
- support for heterogeneous elements



```ruby
# ruby_menagerie.rb

class Duck
  def talk
    puts 'Quack! Quack!'
  end

  def walk
    puts 'Walking like a duck!'
  end
end

class Bird
  def talk
    puts 'Tweet-tweet!'
  end

  def fly
    puts 'Flying like a bird!'
  end
end

class Rabbit
  def jump
    puts 'Jumping like a rabbit!'
  end
end

# Putting a duck, a bird, and a rabbit in an array
menagerie = {"my duck"=>Duck.new, "my bird"=>Bird.new, "my rabbit"=>Rabbit.new}

# Get the duck
duck = menagerie["my duck"] # no need for downcast to a Duck !
duck.talk
duck.walk

# Get the bird
bird = menagerie["my bird"] # no need for downcast to a Bird !
bird.talk
bird.fly

# Get the rabbit
rabbit = menagerie["my rabbit"] # no need for downcast to a Rabbit !
rabbit.jump
```

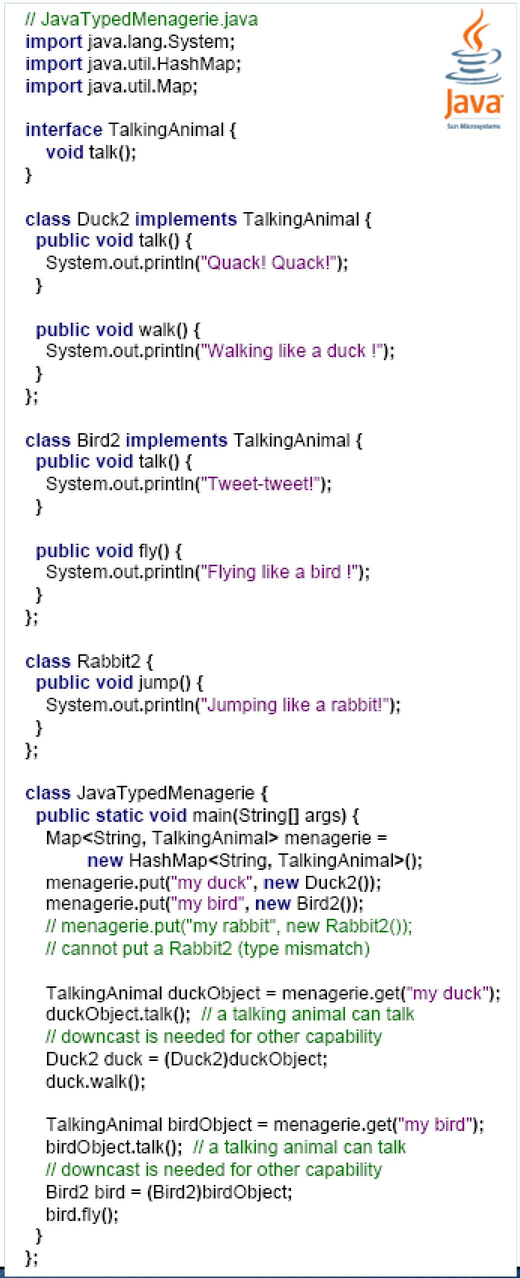
Ruby : a Dynamic Typing Language (4/8)

Java untyped collections support heterogeneous elements but require downcast.



```
// JavaUntypedMenagerie.java
import java.lang.System;
import java.util.HashMap;
import java.util.Map;

class Duck {
  public void talk() {
    System.out.println("Quack! Quack!");
  }

  public void walk() {
    System.out.println("Walking like a duck!");
  }
};

class Bird {
  public void talk() {
    System.out.println("Tweet-tweet!");
  }

  public void fly() {
    System.out.println("Flying like a bird!");
  }
};

class Rabbit {
  public void jump() {
    System.out.println("Jumping like a rabbit!");
  }
};

public class JavaUntypedMenagerie {
  public static void main(String[] args) {
    Map menagerie = new HashMap();
    menagerie.put("my duck", new Duck());
    menagerie.put("my bird", new Bird());
    menagerie.put("my rabbit", new Rabbit());

    Object duckObject = menagerie.get("my duck");
    // downcast is needed !
    Duck duck = (Duck)duckObject;
    duck.talk();
    duck.walk();

    Object birdObject = menagerie.get("my bird");
    // downcast is needed !
    Bird bird = (Bird)birdObject;
    bird.talk();
    bird.fly();

    Object rabbitObject = menagerie.get("my rabbit");
    // downcast is needed !
    Rabbit rabbit = (Rabbit)rabbitObject;
    rabbit.jump();
  }
};
```

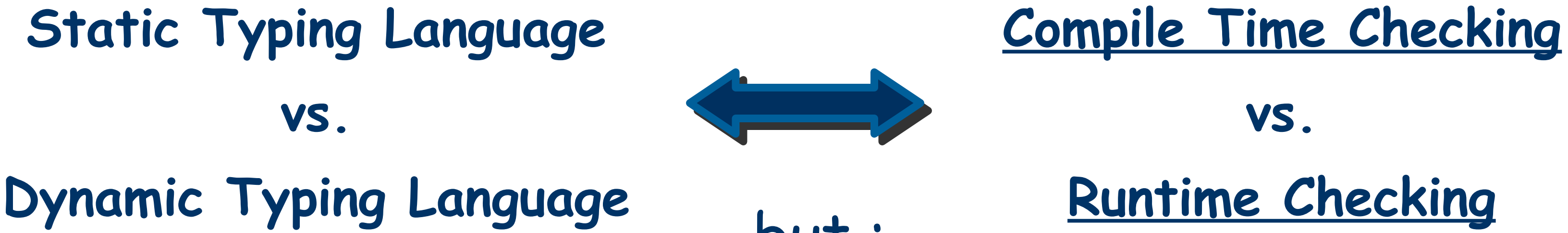
Java typed collections do not support heterogeneous elements but avoid (most) downcast.



```
// JavaTypedMenagerie.java
import java.lang.System;
import java.util.HashMap;
import java.util.Map;

interface TalkingAnimal {
    void talk();
}

class Duck2 implements TalkingAnimal {
  public void talk() {
    System.out.println("Quack! Quack!");
  }

  public void walk() {
    System.out.println("Walking like a duck !");
  }
};

class Bird2 implements TalkingAnimal {
  public void talk() {
    System.out.println("Tweet-tweet!");
  }

  public void fly() {
    System.out.println("Flying like a bird !");
  }
};

class Rabbit2 {
  public void jump() {
    System.out.println("Jumping like a rabbit!");
  }
};

class JavaTypedMenagerie {
  public static void main(String[] args) {
    Map<String, TalkingAnimal> menagerie =
        new HashMap<String, TalkingAnimal>();
    menagerie.put("my duck", new Duck2());
    menagerie.put("my bird", new Bird2());
    // menagerie.put("my rabbit", new Rabbit2());
    // cannot put a Rabbit2 (type mismatch)

    TalkingAnimal duckObject = menagerie.get("my duck");
    duckObject.talk();  // a talking animal can talk
    // downcast is needed for other capability
    Duck2 duck = (Duck2)duckObject;
    duck.walk();

    TalkingAnimal birdObject = menagerie.get("my bird");
    birdObject.talk();  // a talking animal can talk
    // downcast is needed for other capability
    Bird2 bird = (Bird2)birdObject;
    bird.fly();
  }
};
```

Ruby : a Dynamic Typing Language (5/8)

Drawbacks ?



Static Typing Language

vs.

Dynamic Typing Language

<u>Compile Time Checking</u>

vs.

<u>Runtime Checking</u>

but :

- <u>Compiling doesn't mean it executes properly</u>
- The only guarantee of correctness, …, is whether it passes all the tests that define the correctness of your program
- What we need is <u>strong testing, not "strong" typing</u>

"Strong" Typing vs. Strong Testing

(Bruce Eckel, author of "Thinking in C++", "Thinking in Java", "Thinking in Python")

Ruby : a Dynamic Typing Language (6/8)

You can create a large, complex and safe system with a (good) dynamic language.

Example :

(A.K. Erlang but also ERicsson LANGuage)

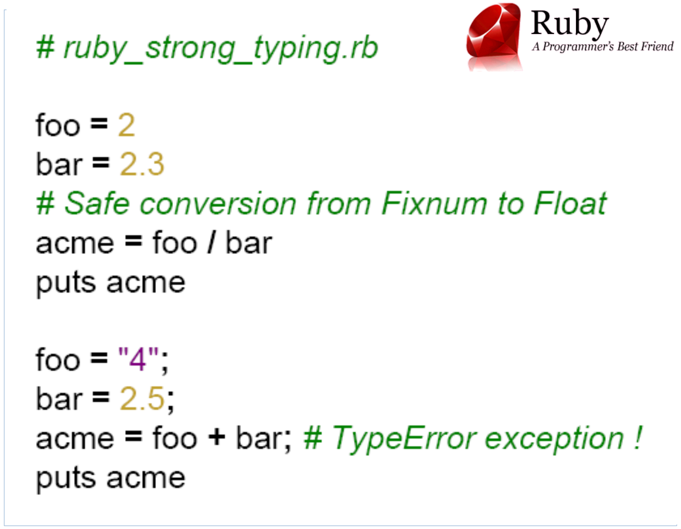
A functional and <u>dynamic typing language</u> designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications.

Ruby : a Dynamic Typing Language (7/8)

Safe Type Conversion

(a.k.a. "Strong Typing")

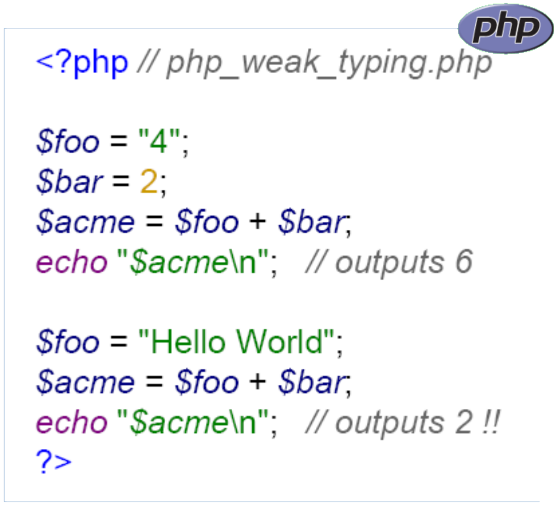
Type Conversion in Ruby is automatically performed only if it is safe.



```
# ruby_strong_typing.rb

foo = 2
bar = 2.3
# Safe conversion from Fixnum to Float
acme = foo / bar
puts acme

foo = "4";
bar = 2.5;
acme = foo + bar; # TypeError exception !
puts acme
```

Conversely, PHP and PERL perform error prone automatic conversion.

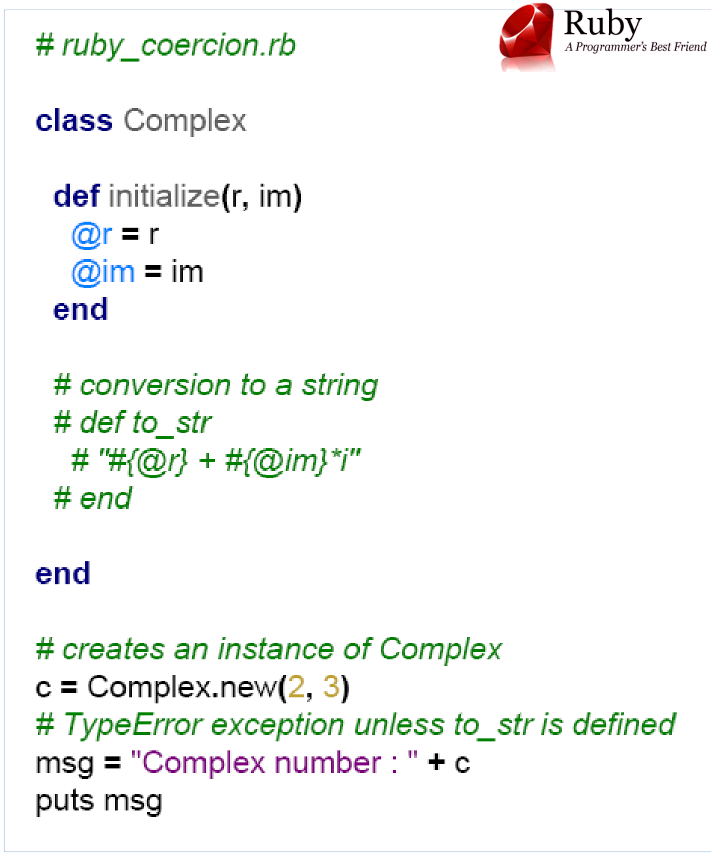
(lack of exception support from day one ? lack of exception culture ?)



```
<?php // php_weak_typing.php

$foo = "4";
$bar = 2;
$acme = $foo + $bar;
echo "$acme\n";   // outputs 6

$foo = "Hello World";
$acme = $foo + $bar;
echo "$acme\n";   // outputs 2 !!
?>
```

Ruby : a Dynamic Typing Language (8/8)

A Ruby custom class may however define a safe conversion (if it really makes sense).



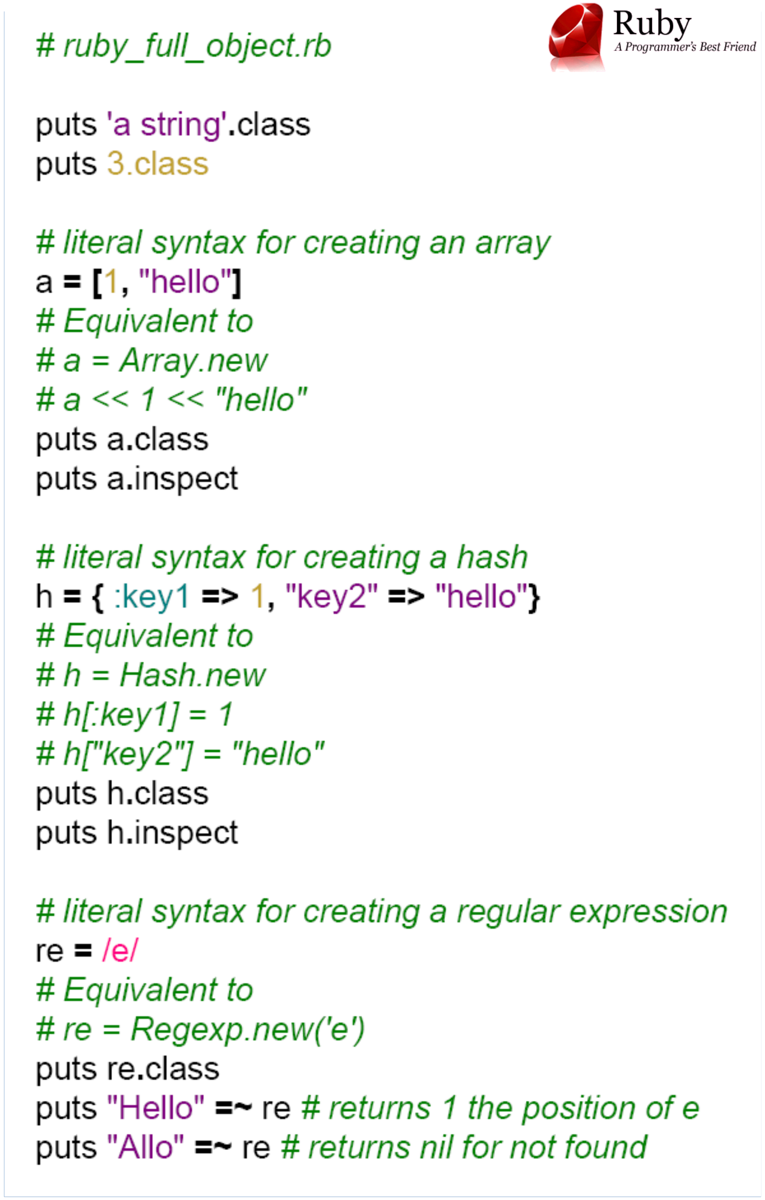
```
# ruby_coercion.rb

class Complex

  def initialize(r, im)
    @r = r
    @im = im
  end

  # conversion to a string
  # def to_str
    # "#{@r} + #{@im}*i"
  # end

end

# creates an instance of Complex
c = Complex.new(2, 3)
# TypeError exception unless to_str is defined
msg = "Complex number : " + c
puts msg
```

Ruby : an Object-Oriented Language

Ruby : an Object-Oriented Language (1/9)

Everything in Ruby is Object.

Built-in classes have a dedicated and friendly syntax ("syntactic sugar").

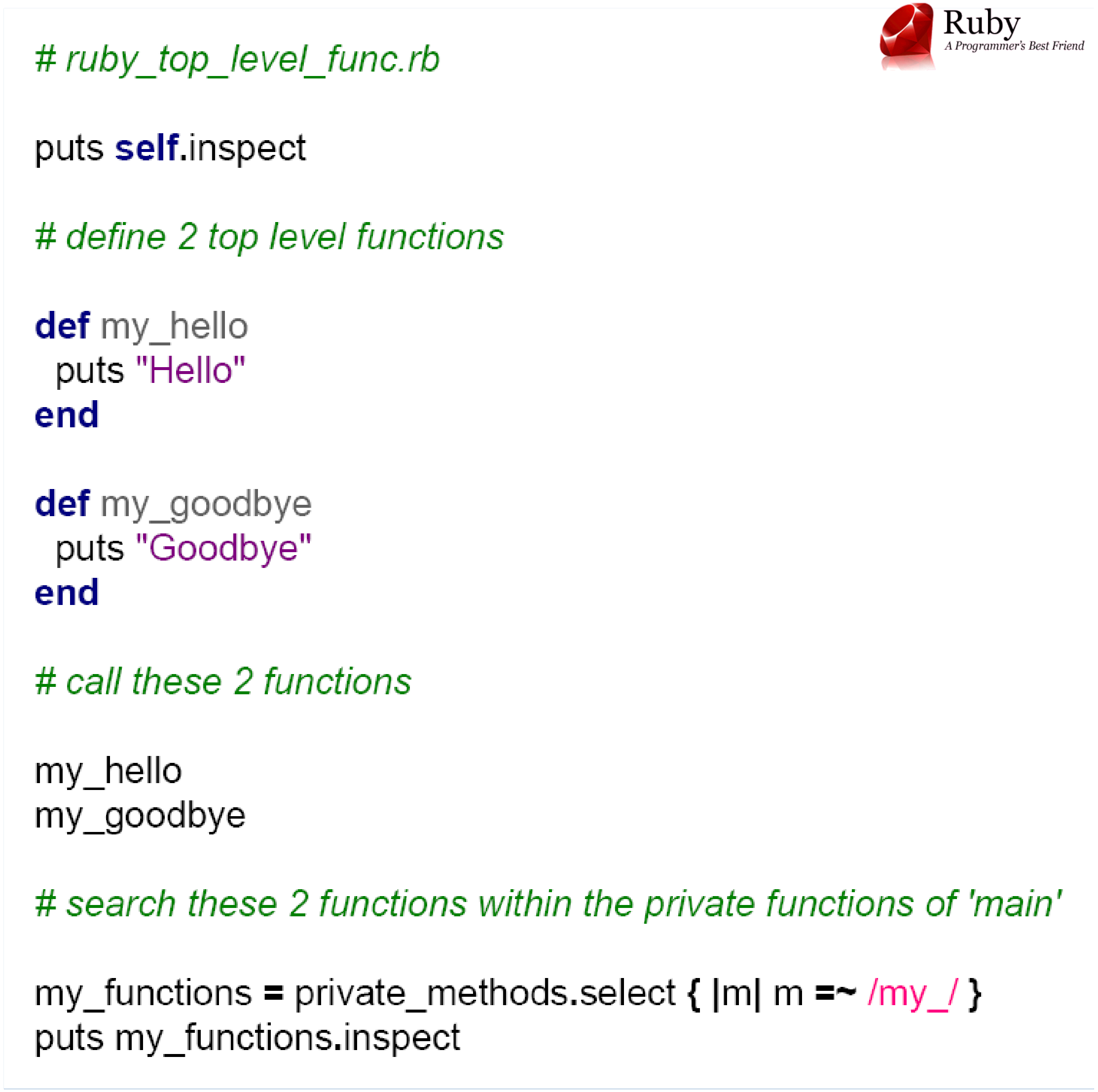


```
# ruby_full_object.rb

puts 'a string'.class
puts 3.class

# literal syntax for creating an array
a = [1, "hello"]
# Equivalent to
# a = Array.new
# a << 1 << "hello"
puts a.class
puts a.inspect

# literal syntax for creating a hash
h = { :key1 => 1, "key2" => "hello"}
# Equivalent to
# h = Hash.new
# h[:key1] = 1
# h["key2"] = "hello"
puts h.class
puts h.inspect

# literal syntax for creating a regular expression
re = /e/
# Equivalent to
# re = Regexp.new('e')
puts re.class
puts "Hello" =~ re # returns 1 the position of e
puts "Allo" =~ re # returns nil for not found
```

Ruby : an Object-Oriented Language (2/9)

Top level functions are in fact private methods of the 'main' object.

Ruby is a fully OO language that can masquerade as a procedural language !



```
# ruby_top_level_func.rb

puts self.inspect

# define 2 top level functions

def my_hello
  puts "Hello"
end

def my_goodbye
  puts "Goodbye"
end

# call these 2 functions

my_hello
my_goodbye

# search these 2 functions within the private functions of 'main'

my_functions = private_methods.select { |m| m =~ /my_/ }
puts my_functions.inspect
```

Ruby : an Object-Oriented Language (3/9)

The primary goal of OO is to reflect real world :

- inheritance : specific / general relationship
- encapsulation : the inside is protected from the outside

Ruby : an Object-Oriented Language (4/9)

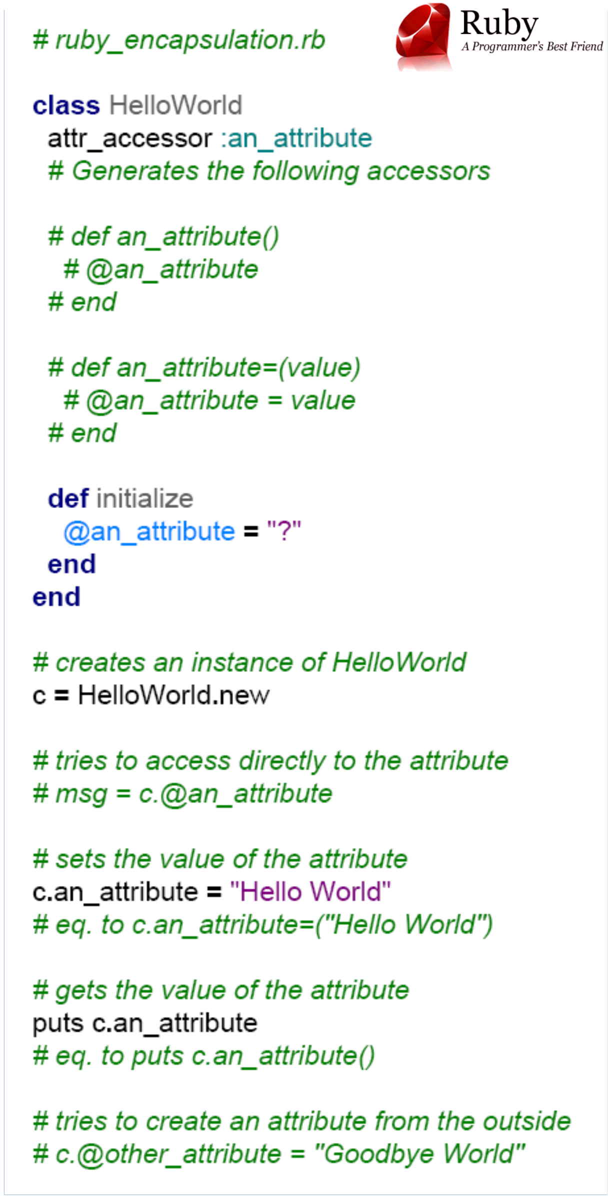
Ruby ensures encapsulation :

- an attribute is always private.

  The access to the attribute are possible only through methods ("accessors").

- an attribute can be defined only within its class definition.

Ruby's accessors look like real attributes (Ruby syntactic sugar again) !

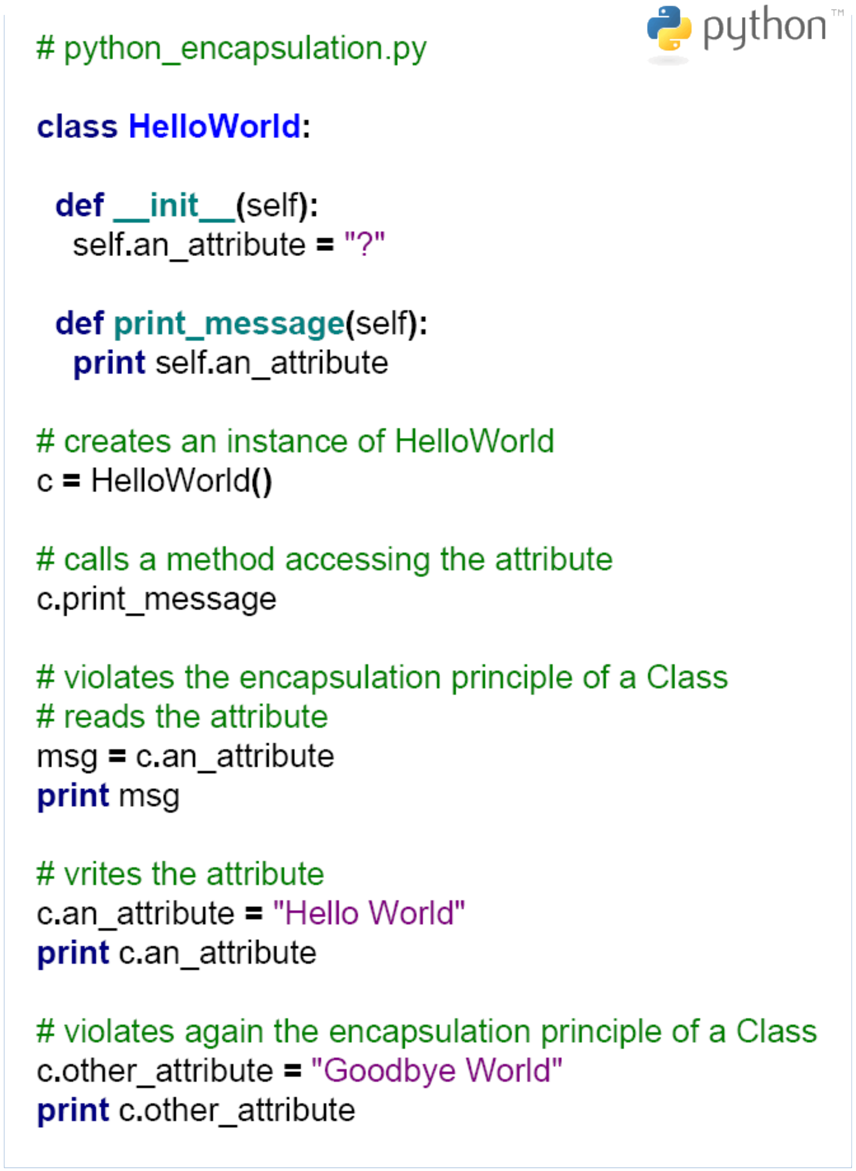


```
# ruby_encapsulation.rb

class HelloWorld
  attr_accessor :an_attribute
  # Generates the following accessors

  # def an_attribute()
    # @an_attribute
  # end

  # def an_attribute=(value)
    # @an_attribute = value
  # end

  def initialize
    @an_attribute = "?"
  end
end

# creates an instance of HelloWorld
c = HelloWorld.new

# tries to access directly to the attribute
# msg = c.@an_attribute

# sets the value of the attribute
c.an_attribute = "Hello World"
# eq. to c.an_attribute=("Hello World")

# gets the value of the attribute
puts c.an_attribute
# eq. to puts c.an_attribute()

# tries to create an attribute from the outside
# c.@other_attribute = "Goodbye World"
```

Ruby : an Object-Oriented Language (5/9)

Python is more lax !

Python has no visibility mechanism !



```
# python_encapsulation.py

class HelloWorld:

 def __init__(self):
  self.an_attribute = "?"

 def print_message(self):
  print self.an_attribute

# creates an instance of HelloWorld
c = HelloWorld()

# calls a method accessing the attribute
c.print_message

# violates the encapsulation principle of a Class
# reads the attribute
msg = c.an_attribute
print msg

# vrites the attribute
c.an_attribute = "Hello World"
print c.an_attribute

# violates again the encapsulation principle of a Class
c.other_attribute = "Goodbye World"
print c.other_attribute
```

Ruby : an Object-Oriented Language (6/9)

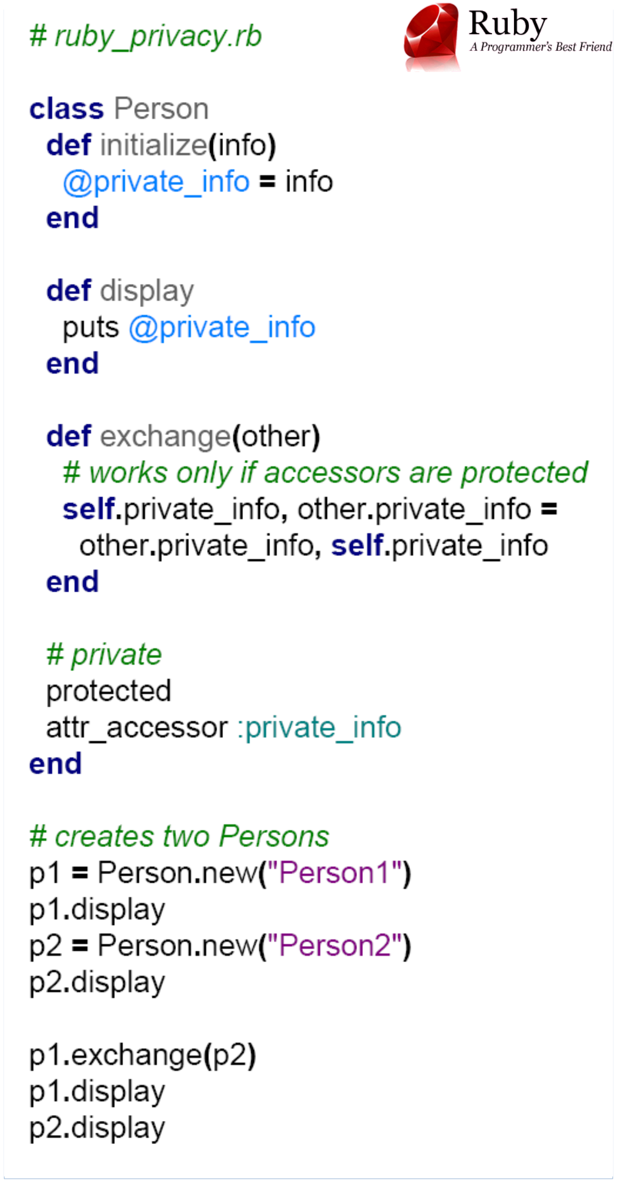
For Ruby (unlike C++ and Java) :

- private really means private

  Another instance of the same class / a derived class cannot access to a private member

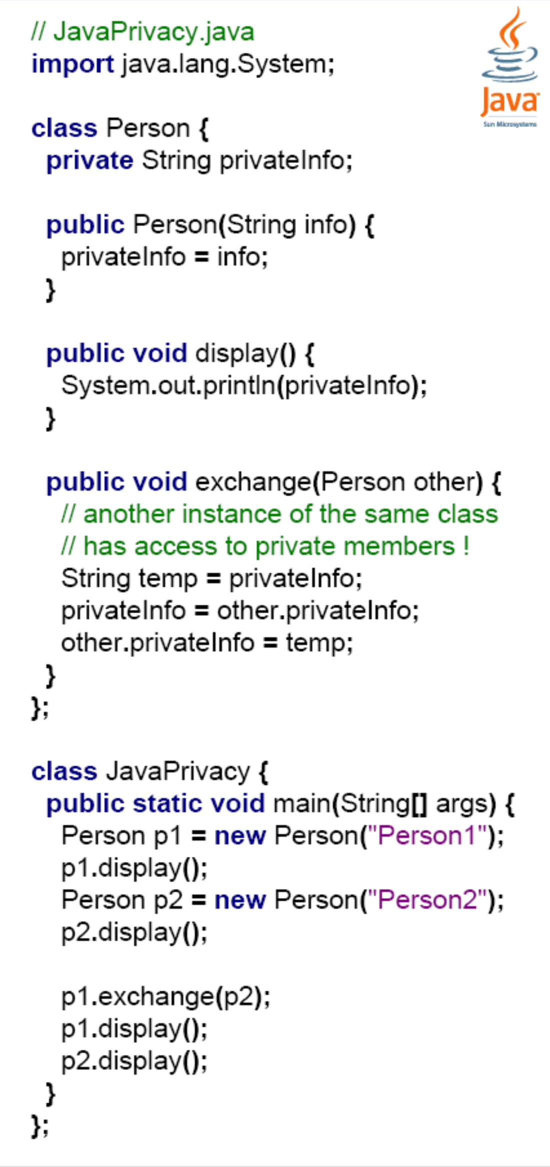
- protected means accessible only within a family

  Another instance of the same class / a derived class can access to a protected member



Ruby
A Programmer's Best Friend

```ruby
# ruby_privacy.rb

class Person
  def initialize(info)
    @private_info = info
  end

  def display
    puts @private_info
  end

  def exchange(other)
    # works only if accessors are protected
    self.private_info, other.private_info =
      other.private_info, self.private_info
  end

  # private
  protected
  attr_accessor :private_info
end

# creates two Persons
p1 = Person.new("Person1")
p1.display
p2 = Person.new("Person2")
p2.display

p1.exchange(p2)
p1.display
p2.display
```



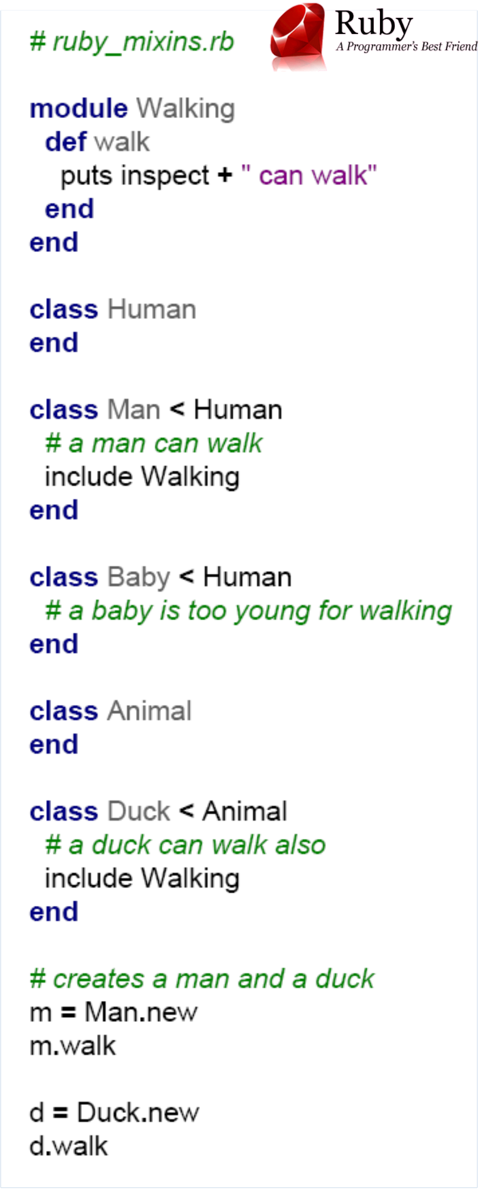
```java
// JavaPrivacy.java
import java.lang.System;

class Person {
  private String privateInfo;

  public Person(String info) {
    privateInfo = info;
  }

  public void display() {
    System.out.println(privateInfo);
  }

  public void exchange(Person other) {
    // another instance of the same class
    // has access to private members !
    String temp = privateInfo;
    privateInfo = other.privateInfo;
    other.privateInfo = temp;
  }
};

class JavaPrivacy {
  public static void main(String[] args) {
    Person p1 = new Person("Person1");
    p1.display();
    Person p2 = new Person("Person2");
    p2.display();

    p1.exchange(p2);
    p1.display();
    p2.display();
  }
};
```

Ruby : an Object-Oriented Language (7/9)

Java's single inheritance is annoying : reusing code from another class requires adapter code.

C++ multiple inheritance is powerful but is very complex.

Ruby's mix-in feature provides the power of multiple inheritance without its complexity.



```
# ruby_mixins.rb

module Walking
  def walk
    puts inspect + " can walk"
  end
end

class Human
end

class Man < Human
  # a man can walk
  include Walking
end

class Baby < Human
  # a baby is too young for walking
end

class Animal
end

class Duck < Animal
  # a duck can walk also
  include Walking
end

# creates a man and a duck
m = Man.new
m.walk

d = Duck.new
d.walk
```

Ruby : an Object-Oriented Language (8/9)

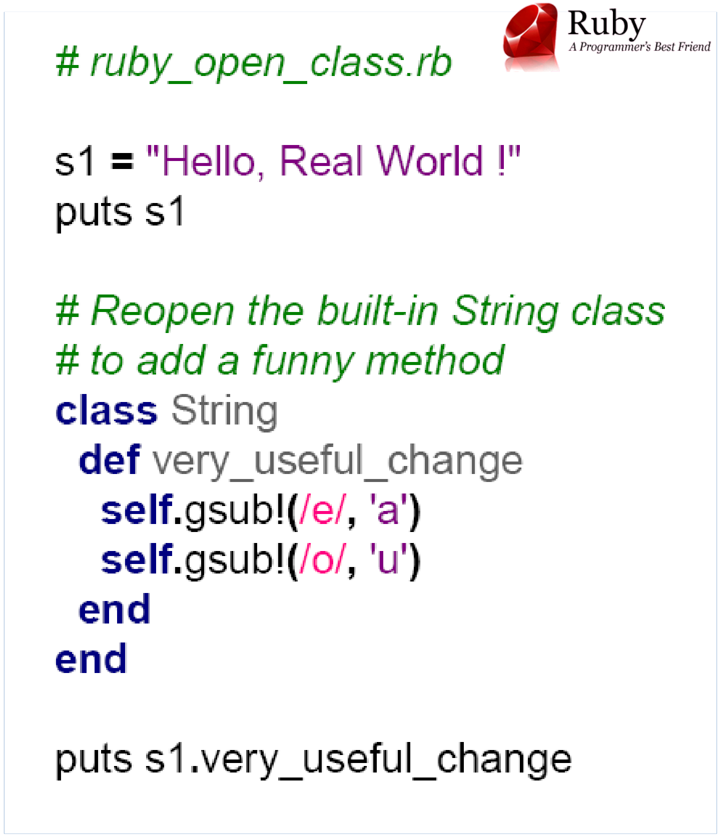
You're not forced to derive a class just to extend its capabilities.

You can reopen it !

Benefits :

You use naturally the same class.

Useful also if you do not control how objects are created (you cannot instantiate a derived class instead of the base class).

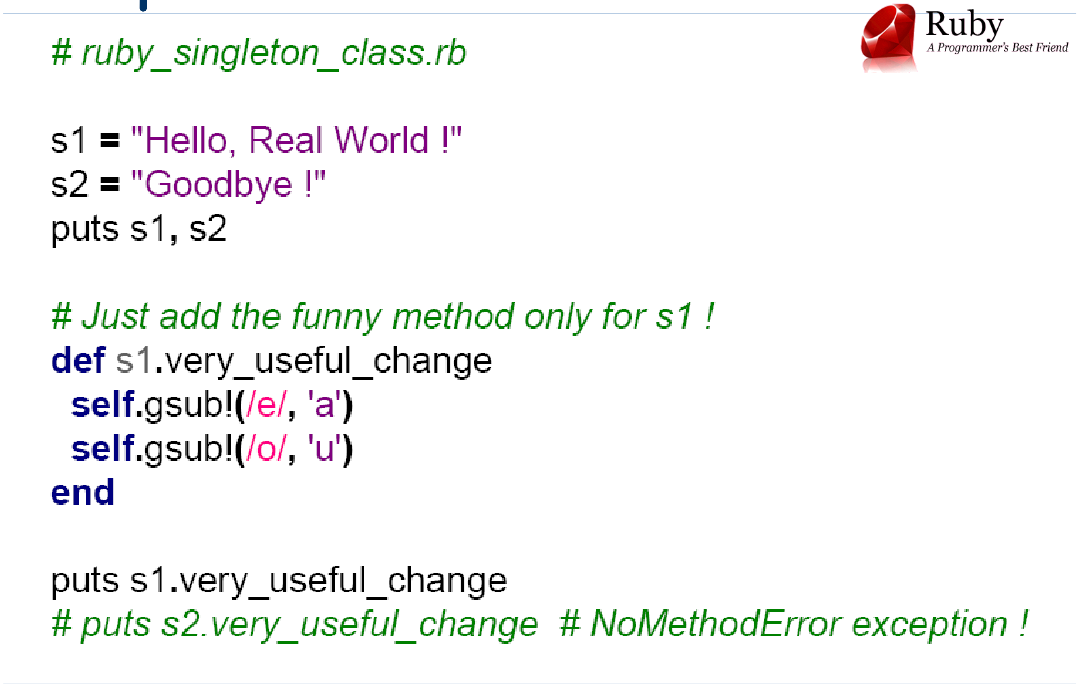


```
# ruby_open_class.rb

s1 = "Hello, Real World !"
puts s1

# Reopen the built-in String class
# to add a funny method
class String
  def very_useful_change
    self.gsub!(/e/, 'a')
    self.gsub!(/o/, 'u')
  end
end

puts s1.very_useful_change
```

Ruby : an Object-Oriented Language (9/9)

If you reopen a class and add methods to it, all existing instances will benefit from them.

But you can also just add methods to a given instance if you don't want to impact other instances.



```ruby
# ruby_singleton_class.rb

s1 = "Hello, Real World !"
s2 = "Goodbye !"
puts s1, s2

# Just add the funny method only for s1 !
def s1.very_useful_change
  self.gsub!(/e/, 'a')
  self.gsub!(/o/, 'u')
end

puts s1.very_useful_change
# puts s2.very_useful_change  # NoMethodError exception !
```

DRb (Distributed Ruby) uses this feature to indicate whether an object will be transmitted by value or by reference (through a module inclusion).

Ruby : a good taste of Functional Programming

Functional Programming - Principles

Functional programming languages are a class of languages designed to reflect the way people think mathematically, rather than reflecting the underlying machine. ***[Goldberg]***

"Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands.

The expressions in these language are formed by using functions to combine basic values.

A functional language is a language that supports and encourages programming in a functional style." ***[comp.lang.functional FAQ]***

"A functional language does not allow any destructive

operation — one which overwrites data — such as assignment.

Purely functional languages are free of side effects, i.e.,

invoking a function has no effect other than computing the value returned by the function." ***[NIST]***

Functional Programming - Principles

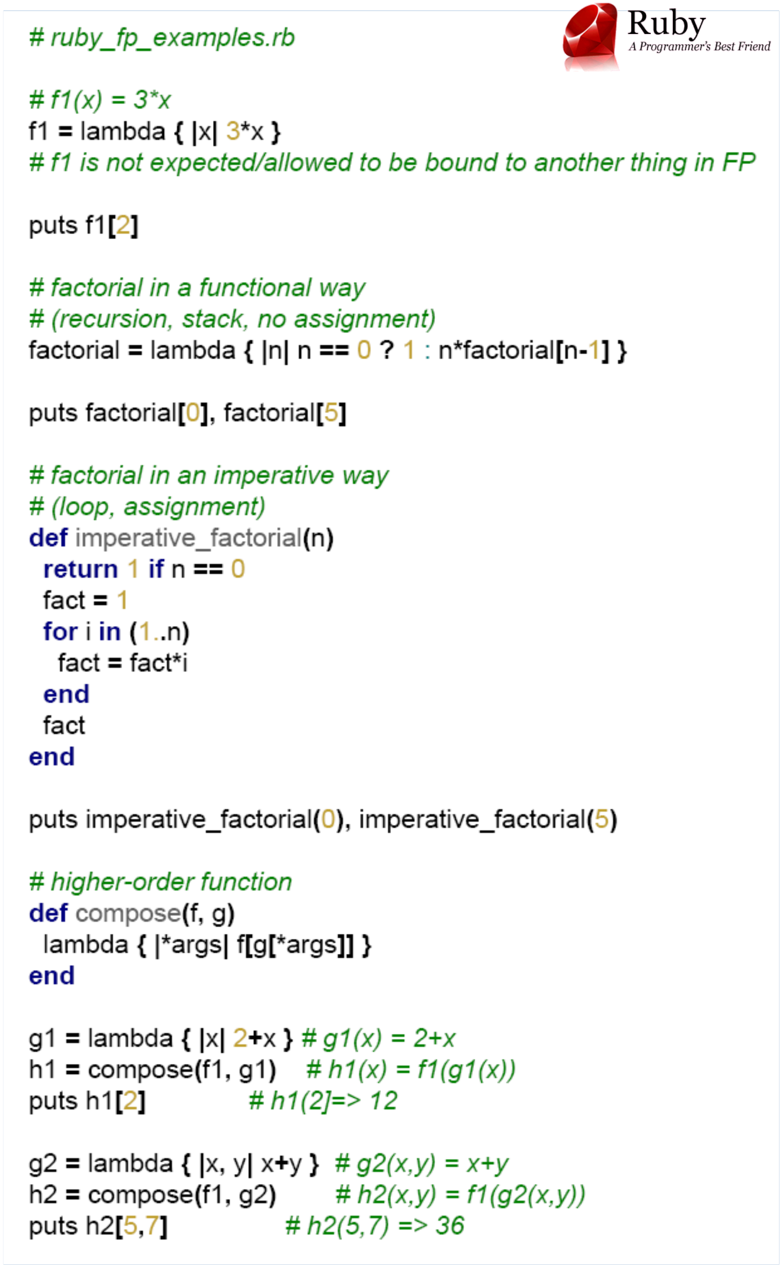
- Every symbol is final in (pure) Functional Programming

  x = f(y) just means wherever you have x, you can replace it with f(y) and vice-versa.

- Repetition is expressed via recursion.
- Higher-Order Function : a function that takes / returns functions as parameters
- Stack is the rule (over Heap).

Benefits :

- Unit Testing is easier (no side-effects)
- Concurrency is provided as free (e.g. ERLANG)



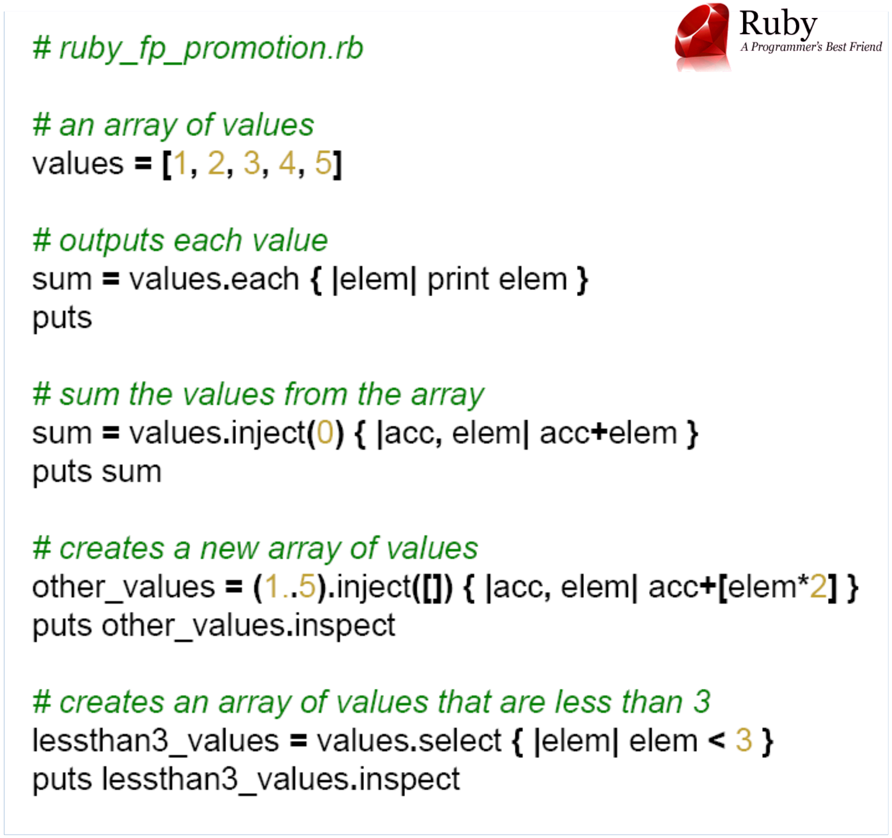
```
# ruby_fp_examples.rb

# f1(x) = 3*x
f1 = lambda { |x| 3*x }
# f1 is not expected/allowed to be bound to another thing in FP

puts f1[2]

# factorial in a functional way
# (recursion, stack, no assignment)
factorial = lambda { |n| n == 0 ? 1 : n*factorial[n-1] }

puts factorial[0], factorial[5]

# factorial in an imperative way
# (loop, assignment)
def imperative_factorial(n)
  return 1 if n == 0
  fact = 1
  for i in (1..n)
    fact = fact*i
  end
  fact
end

puts imperative_factorial(0), imperative_factorial(5)

# higher-order function
def compose(f, g)
  lambda { |*args| f[g[*args]] }
end

g1 = lambda { |x| 2+x } # g1(x) = 2+x
h1 = compose(f1, g1)    # h1(x) = f1(g1(x))
puts h1[2]              # h1(2]=> 12

g2 = lambda { |x, y| x+y }  # g2(x,y) = x+y
h2 = compose(f1, g2)        # h2(x,y) = f1(g2(x,y))
puts h2[5,7]                # h2(5,7) => 36
```

Functional Programming - Ruby Blocks

Ruby is not a (pure) Functional Programming Language but it favours Functional Programming.

Ruby methods notably from the Enumerable module take an anonymous function as a parameter : a Ruby block.

You can write Functional Programs in Ruby if you avoid side-effects.



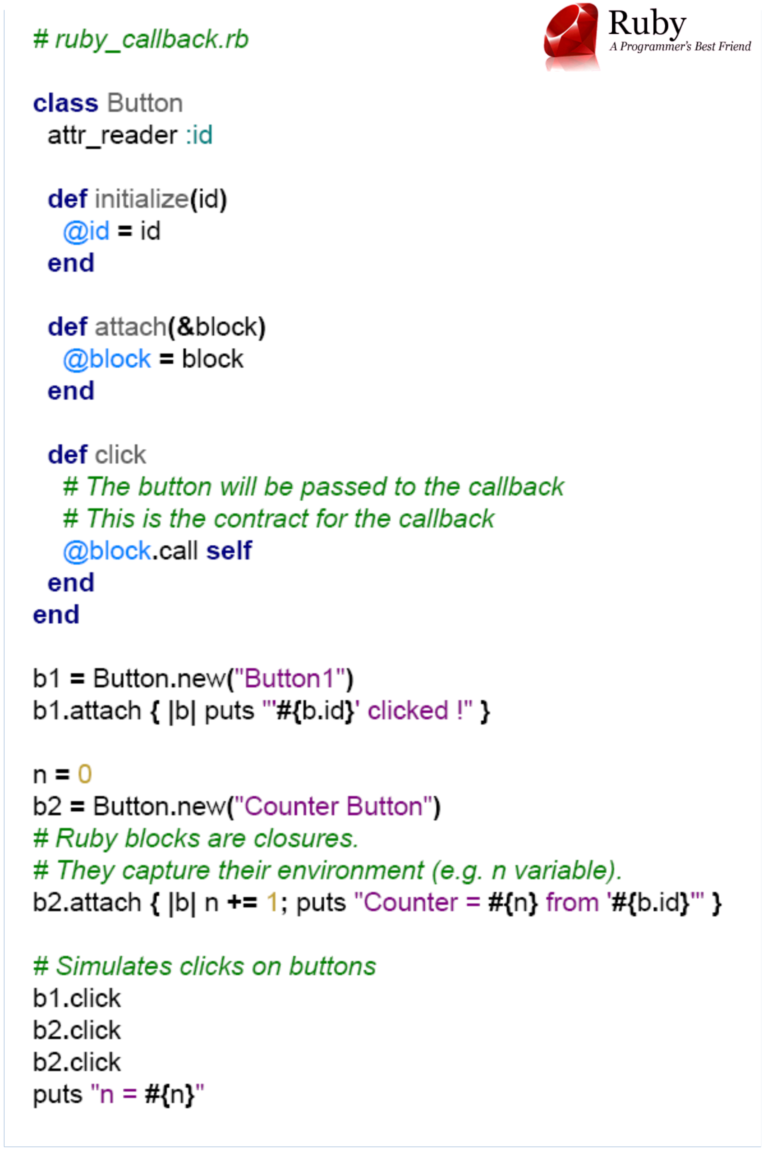
```
# ruby_fp_promotion.rb

# an array of values
values = [1, 2, 3, 4, 5]

# outputs each value
sum = values.each { |elem| print elem }
puts

# sum the values from the array
sum = values.inject(0) { |acc, elem| acc+elem }
puts sum

# creates a new array of values
other_values = (1..5).inject([]) { |acc, elem| acc+[elem*2] }
puts other_values.inspect

# creates an array of values that are less than 3
lessthan3_values = values.select { |elem| elem < 3 }
puts lessthan3_values.inspect
```

Note : A pure Functional Language must interface with the "real" side-effects world for Graphics, Input/Output.

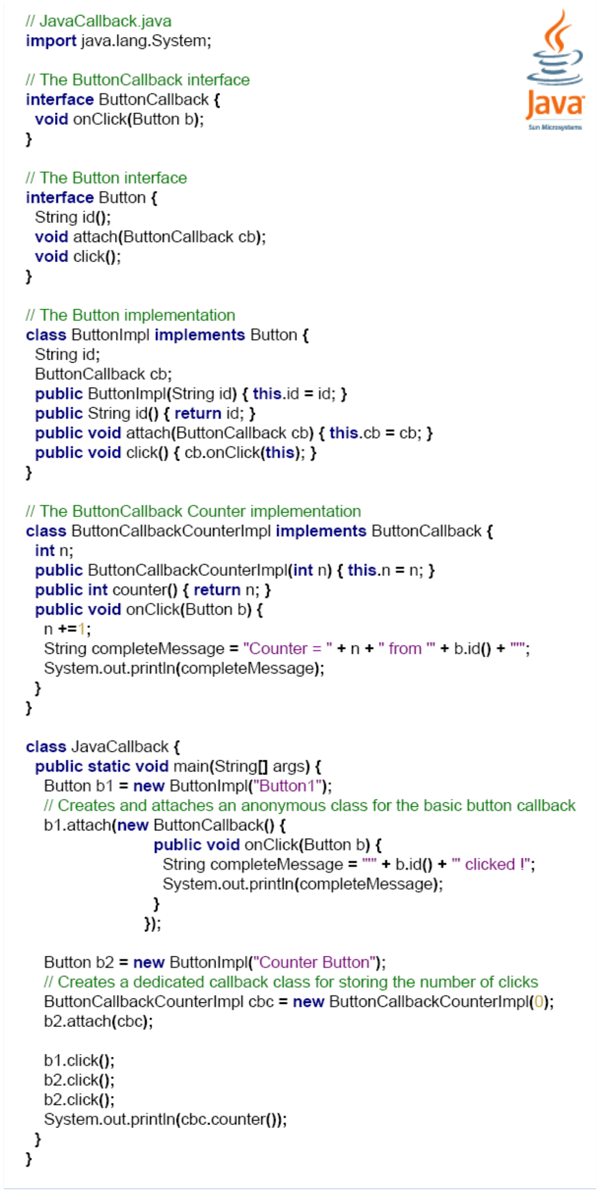
Functional Programming - Ruby Blocks

Ruby Blocks are much easier than Java classes for implementing callbacks.

Ruby Blocks are real closures : they capture their context. They allow easy communication between the block and its context.



```ruby
# ruby_callback.rb

class Button
  attr_reader :id

  def initialize(id)
    @id = id
  end

  def attach(&block)
    @block = block
  end

  def click
    # The button will be passed to the callback
    # This is the contract for the callback
    @block.call self
  end
end

b1 = Button.new("Button1")
b1.attach { |b| puts "'#{b.id}' clicked !" }

n = 0
b2 = Button.new("Counter Button")
# Ruby blocks are closures.
# They capture their environment (e.g. n variable).
b2.attach { |b| n += 1; puts "Counter = #{n} from '#{b.id}'" }

# Simulates clicks on buttons
b1.click
b2.click
b2.click
puts "n = #{n}"
```



```java
// JavaCallback.java
import java.lang.System;

// The ButtonCallback interface
interface ButtonCallback {
 void onClick(Button b);
}

// The Button interface
interface Button {
 String id();
 void attach(ButtonCallback cb);
 void click();
}

// The Button implementation
class ButtonImpl implements Button {
 String id;
 ButtonCallback cb;
 public ButtonImpl(String id) { this.id = id; }
 public String id() { return id; }
 public void attach(ButtonCallback cb) { this.cb = cb; }
 public void click() { cb.onClick(this); }
}

// The ButtonCallback Counter implementation
class ButtonCallbackCounterImpl implements ButtonCallback {
 int n;
 public ButtonCallbackCounterImpl(int n) { this.n = n; }
 public int counter() { return n; }
 public void onClick(Button b) {
  n +=1;
  String completeMessage = "Counter = " + n + " from '" + b.id() + "'";
  System.out.println(completeMessage);
 }
}

class JavaCallback {
 public static void main(String[] args) {
  Button b1 = new ButtonImpl("Button1");
  // Creates and attaches an anonymous class for the basic button callback
  b1.attach(new ButtonCallback() {
                public void onClick(Button b) {
                 String completeMessage = "'" + b.id() + "' clicked !";
                 System.out.println(completeMessage);
                }
               });

  Button b2 = new ButtonImpl("Counter Button");
  // Creates a dedicated callback class for storing the number of clicks
  ButtonCallbackCounterImpl cbc = new ButtonCallbackCounterImpl(0);
  b2.attach(cbc);

  b1.click();
  b2.click();
  b2.click();
  System.out.println(cbc.counter());
 }
}
```

Ruby : a highly Reflective Language

Terminology

Meta-programming :

The writing of programs that write or manipulate other programs (or themselves).

Benefit : Less code is written manually.

Meta-language

The language in which the meta-program is written.

Reflective Language

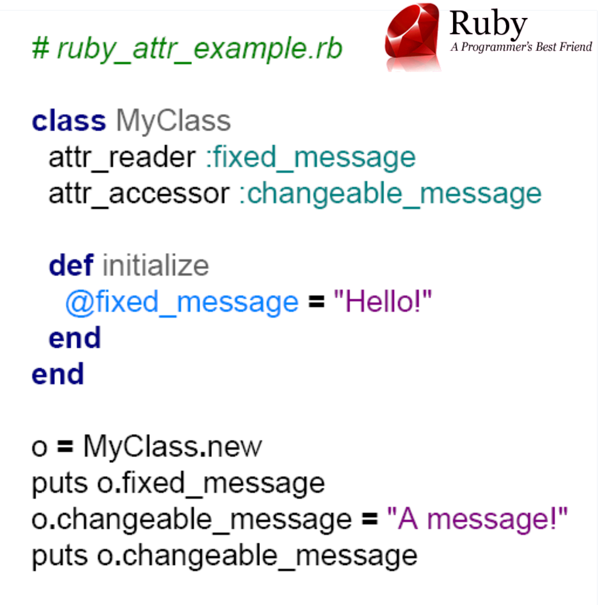
A programming language whose meta-language is itself.

Meta-programming & Encapsulation :

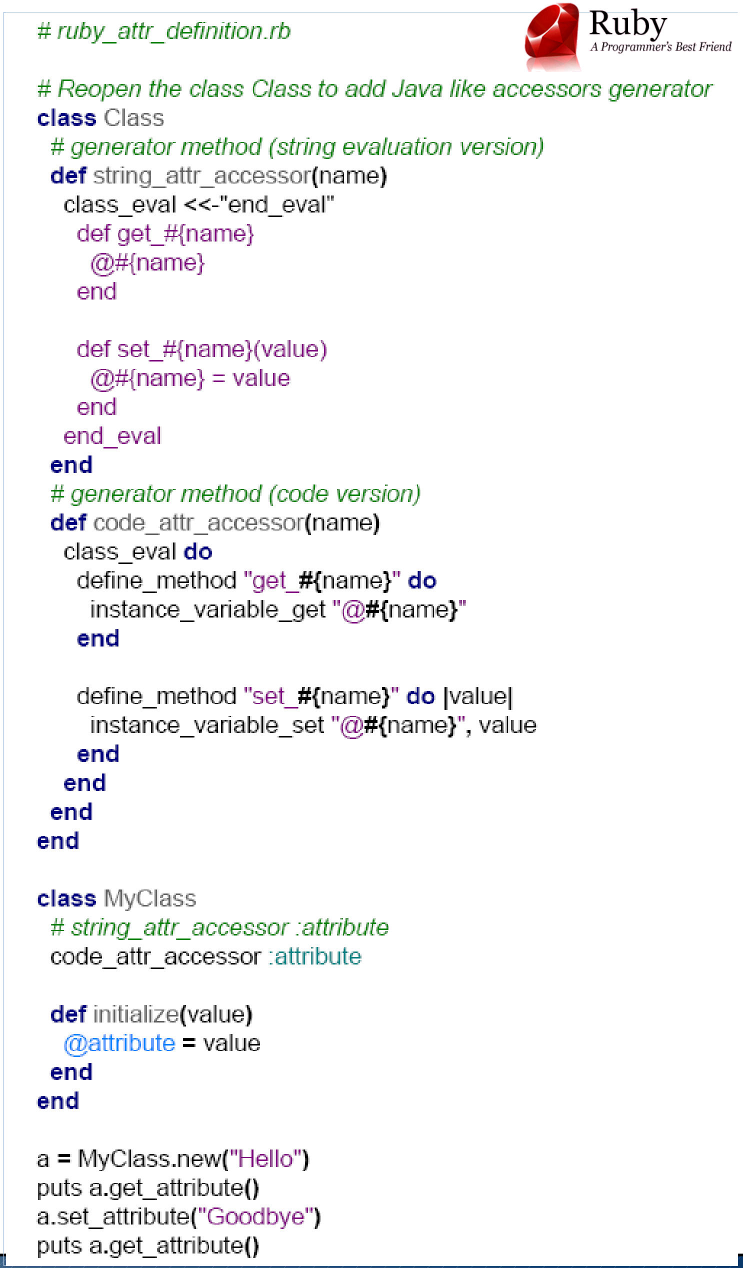
- You're normally force to follow encapsulation. This is the normal and preferred way.
- But if you have a good reason, you can break encapsulation with Ruby meta-programming features.

Ruby Meta-programming

The classical example of Ruby meta-programming : attr_reader, attr_writer, attr_accessor.
These are just class methods that generate respectively read, write, both read and write accessors for a given instance variable.



```ruby
# ruby_attr_example.rb

class MyClass
  attr_reader :fixed_message
  attr_accessor :changeable_message

  def initialize
    @fixed_message = "Hello!"
  end
end

o = MyClass.new
puts o.fixed_message
o.changeable_message = "A message!"
puts o.changeable_message
```



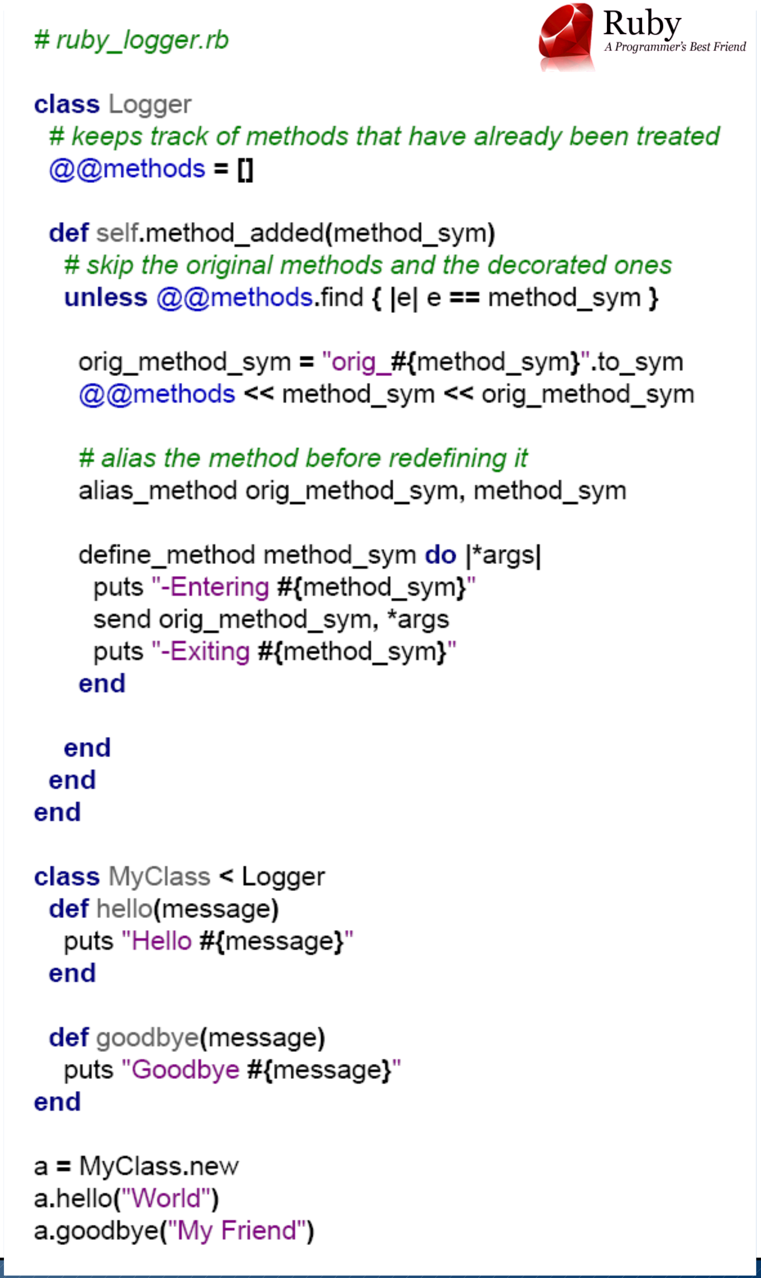
```ruby
# ruby_attr_definition.rb

# Reopen the class Class to add Java like accessors generator
class Class
 # generator method (string evaluation version)
 def string_attr_accessor(name)
  class_eval <<-"end_eval"
   def get_#{name}
    @#{name}
   end

   def set_#{name}(value)
    @#{name} = value
   end
  end_eval
 end
 # generator method (code version)
 def code_attr_accessor(name)
  class_eval do
   define_method "get_#{name}" do
    instance_variable_get "@#{name}"
   end

   define_method "set_#{name}" do |value|
    instance_variable_set "@#{name}", value
   end
  end
 end
end

class MyClass
 # string_attr_accessor :attribute
 code_attr_accessor :attribute

 def initialize(value)
  @attribute = value
 end
end

a = MyClass.new("Hello")
puts a.get_attribute()
a.set_attribute("Goodbye")
puts a.get_attribute()
```

Ruby Meta-programming

You can also generate code automatically upon some events :

- a missing method is called on an object
- a module is being mixed-in
- a class is being inherited
- a method is being added, removed
- …

AOP (Aspect Oriented Programming) can be implemented straightforwardly in Ruby ! (aspectr).



Ruby
A Programmer's Best Friend

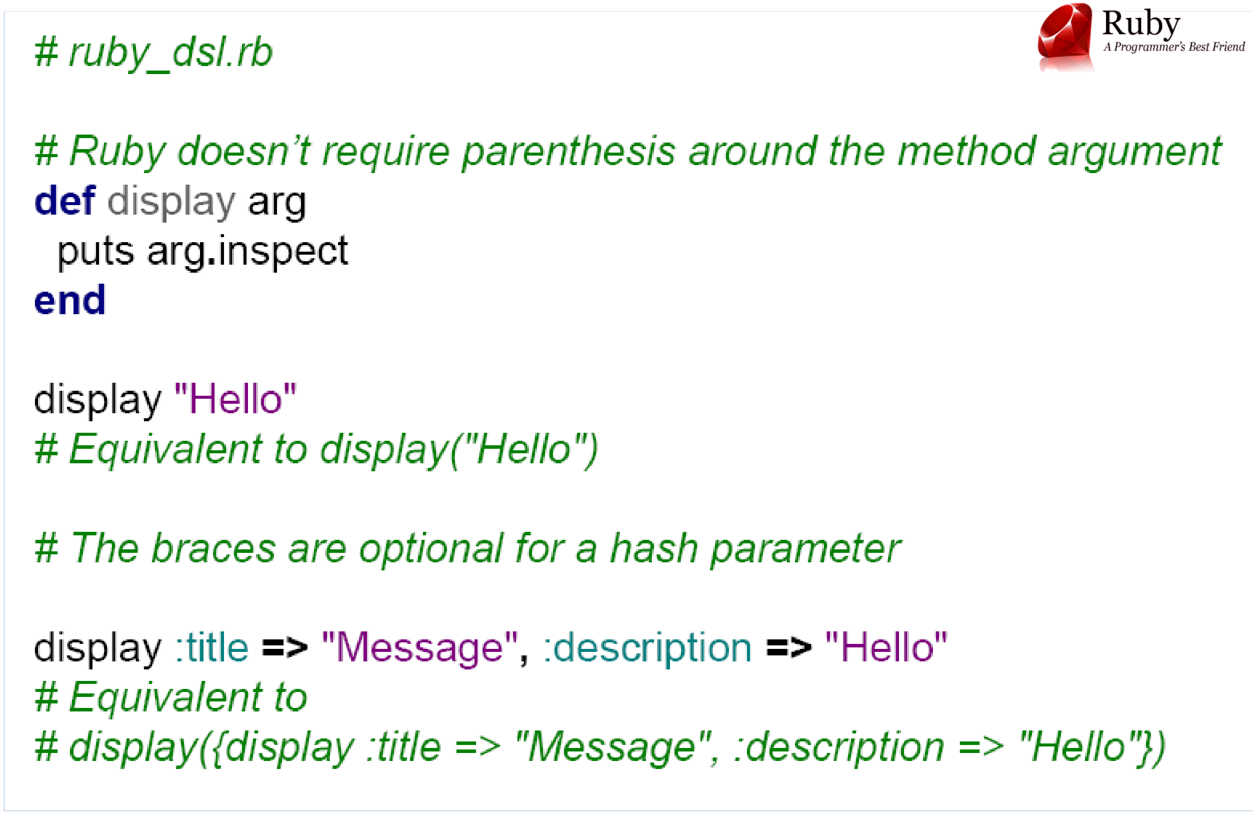
```ruby
# ruby_logger.rb

class Logger
  # keeps track of methods that have already been treated
  @@methods = []

  def self.method_added(method_sym)
    # skip the original methods and the decorated ones
    unless @@methods.find { |e| e == method_sym }

      orig_method_sym = "orig_#{method_sym}".to_sym
      @@methods << method_sym << orig_method_sym

      # alias the method before redefining it
      alias_method orig_method_sym, method_sym

      define_method method_sym do |*args|
        puts "-Entering #{method_sym}"
        send orig_method_sym, *args
        puts "-Exiting #{method_sym}"
      end

    end
  end
end

class MyClass < Logger
  def hello(message)
    puts "Hello #{message}"
  end

  def goodbye(message)
    puts "Goodbye #{message}"
end

a = MyClass.new
a.hello("World")
a.goodbye("My Friend")
```

Ruby : a base for creating DSL

Ruby : a base for creating DSL (1/3)

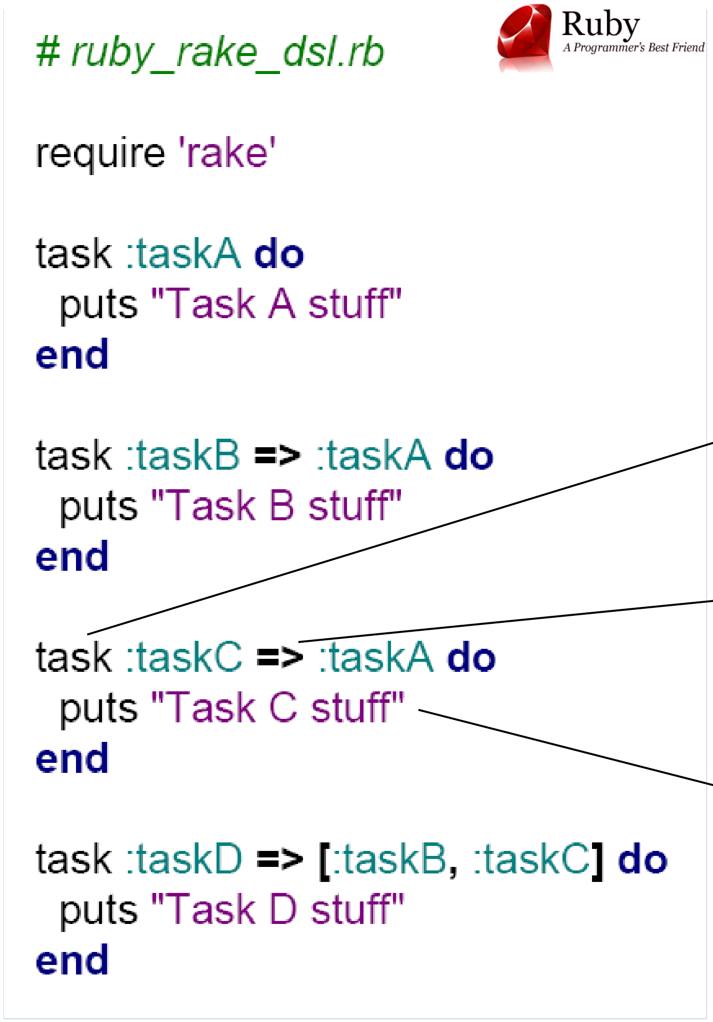
Syntax does matter !

Ruby's syntactic sugar is one of the key point that makes DSL in Ruby very easy.



```
# ruby_dsl.rb

# Ruby doesn’t require parenthesis around the method argument
def display arg
  puts arg.inspect
end

display "Hello"
# Equivalent to display("Hello")

# The braces are optional for a hash parameter

display :title => "Message", :description => "Hello"
# Equivalent to
# display({display :title => "Message", :description => "Hello"})
```

Ruby : a base for creating DSL (2/3)



```
# ruby_rake_dsl.rb

require 'rake'

task :taskA do
  puts "Task A stuff"
end

task :taskB => :taskA do
  puts "Task B stuff"
end

task :taskC => :taskA do
  puts "Task C stuff"
end

task :taskD => [:taskB, :taskC] do
  puts "Task D stuff"
end
```

Example of DSL in Ruby : Rake

A Build Language written in Ruby.

Benefits :

- Readable syntax.
- Full access to the power of Ruby.

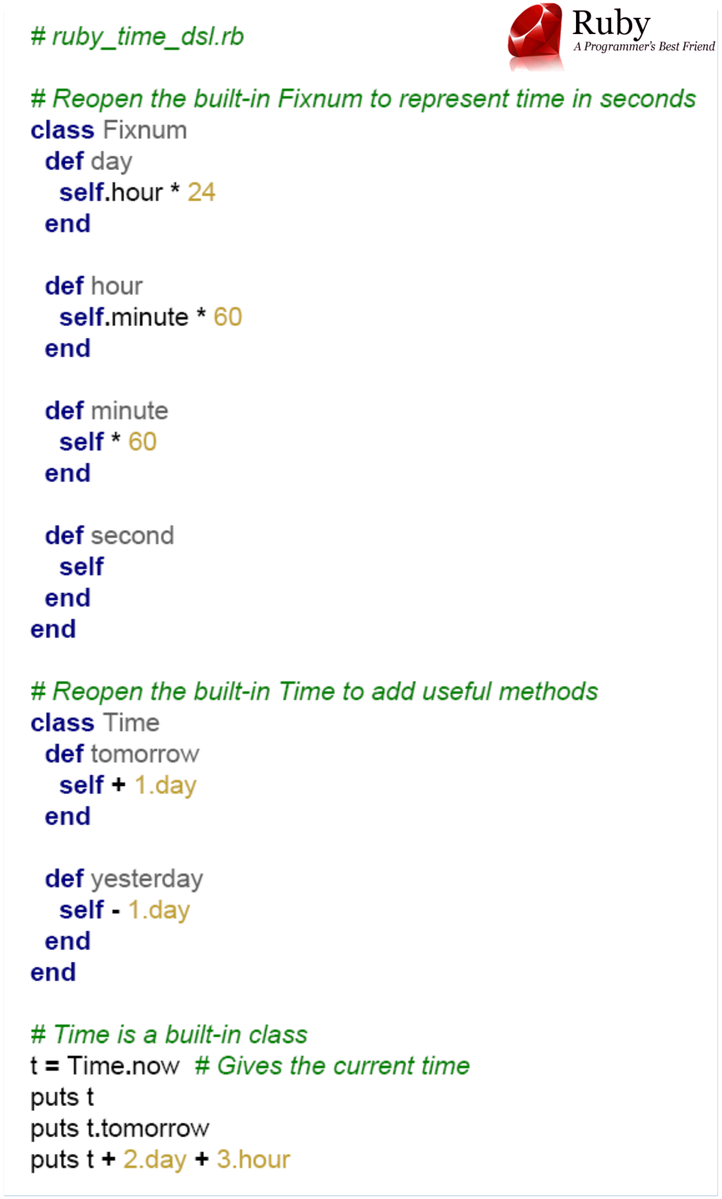
task looks like a keyword but is just a regular Ruby method

the task name (:taskC) and the task requisites (:taskA) are just defined by a hash

What the task has to do is just defined by a Ruby block

Ruby : a base for creating DSL (3/3)

Ruby open class is another key point for DSL.



```
# ruby_time_dsl.rb

# Reopen the built-in Fixnum to represent time in seconds
class Fixnum
  def day
    self.hour * 24
  end

  def hour
    self.minute * 60
  end

  def minute
    self * 60
  end

  def second
    self
  end
end

# Reopen the built-in Time to add useful methods
class Time
  def tomorrow
    self + 1.day
  end

  def yesterday
    self - 1.day
  end
end

# Time is a built-in class
t = Time.now  # Gives the current time
puts t
puts t.tomorrow
puts t + 2.day + 3.hour
```

More Ruby vs. C++, Java, Python, Groovy, PHP

More Ruby vs. C++

- C++ is statically typed,
- C++ is complex,
- C++ meta-programming is only static (template).
- C++ doesn't have introspection.

Conversely, Ruby has garbage collection but is slower.

More Ruby vs. Java

- Java is statically typed.
- Java supports only single inheritance.
- Java meta-programming is only static (before class loading and through Javassist) but Java supports introspection.
- Java is too verbose.
- Java doesn't have neither (yet) closure nor favours Functional Programming.

More Ruby vs. Python

Python has similar capabilities as Ruby but :

- is less Object-Oriented (e.g. encapsulation)
- lacks uniformity (e.g. function vs. method)
- meta-programming is less favoured / natural
- doesn't enable DSL creation

Python has a different philosophy :

"There is only one way to do it"

More Ruby vs. PHP

PHP (PHP: Hypertext Preprocessor) :

A reflective and dynamic programming language originally designed for producing dynamic Web pages and remote application software.

Drawbacks :

- OO has been added lately and is still not yet complete (no class method …)
- The library is procedural !
- No namespace support ! Everything is in the global space !

Ruby vs. Groovy

Groovy has been created to add dynamic-style language features on top of Java.

Heavily influenced by Ruby !

Not (yet?) as powerful as Ruby (open class, …).

Syntax simplification limited to remain close to Java.

It seems to be too late for Groovy :

- Charles Nutter (JRuby core developer) : "we believe Ruby is a better language than we could design ourselves (or design based on Java with dynamic language features) and so we aim to support pure Ruby as closely as possible"
- Ruby has now a greater community and audience (conferences, library of books, …)
- Ruby is supported by SUN (through JRuby)

More on Ruby

Current and future Ruby

The current official implementation (1.8.x) :

- an interpreter
- green threads

The Ruby 2.0 official implementation -
YARV (Yet Another Ruby Virtual Machine) :

- a Virtual Machine with specific Ruby byte-code
- native threads

Current measures : 3.5x faster than interpreter version.

Ruby 1.9 expected for Christmas 2007.

JRuby

An implementation of Ruby 1.8.x on the Java Virtual Machine :

- speed of the Java VM
- native threads

- Benefit : JRuby provides the access of Java platform and libraries to Ruby.
- Drawback : YARV will likely be more effective than JRuby.

Integration

Ruby is for Java what Java is for C/C++ !

There are good Java, C/C++ frameworks/libraries.

Ruby typically wraps and/or integrates these technologies (e.g. JRuby, RubySQLite, …).

That's all folks !