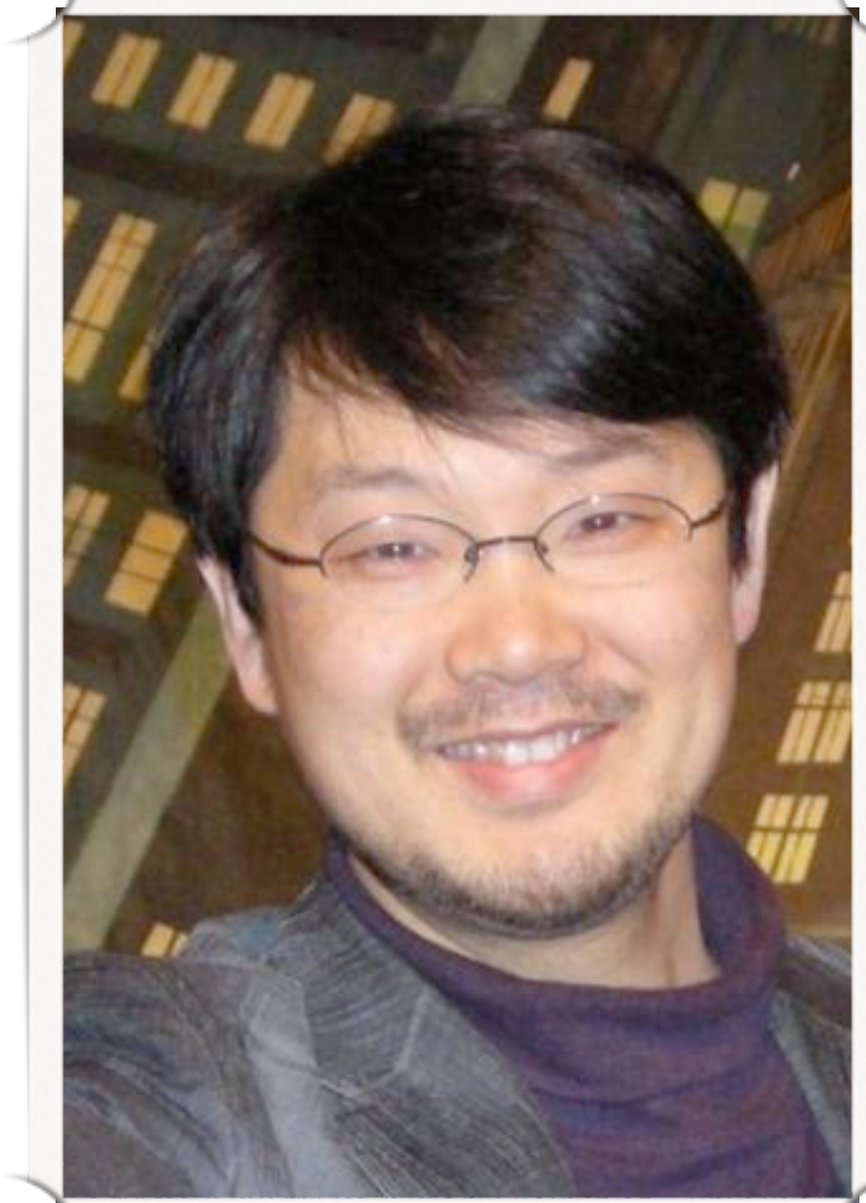
Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

Raúl Montes T.



Yukihiro Matsumoto

“Matz”

1995

"... trying to make Ruby natural, not simple"

"Ruby is simple in appearance, but is very complex inside, just like our human body"

Ruby es flexible

No necesitas terminar las sentencias con “**;**”, pero puedes...

No tienes que encerrar entre **( )** los argumentos, pero puedes...

Puedes redefinir todo el lenguaje si quieres...

Ruby is a **dinamycally** typed language

Ruby is a **strongly** typed language

Las variables no se declaran, simplemente se asignan

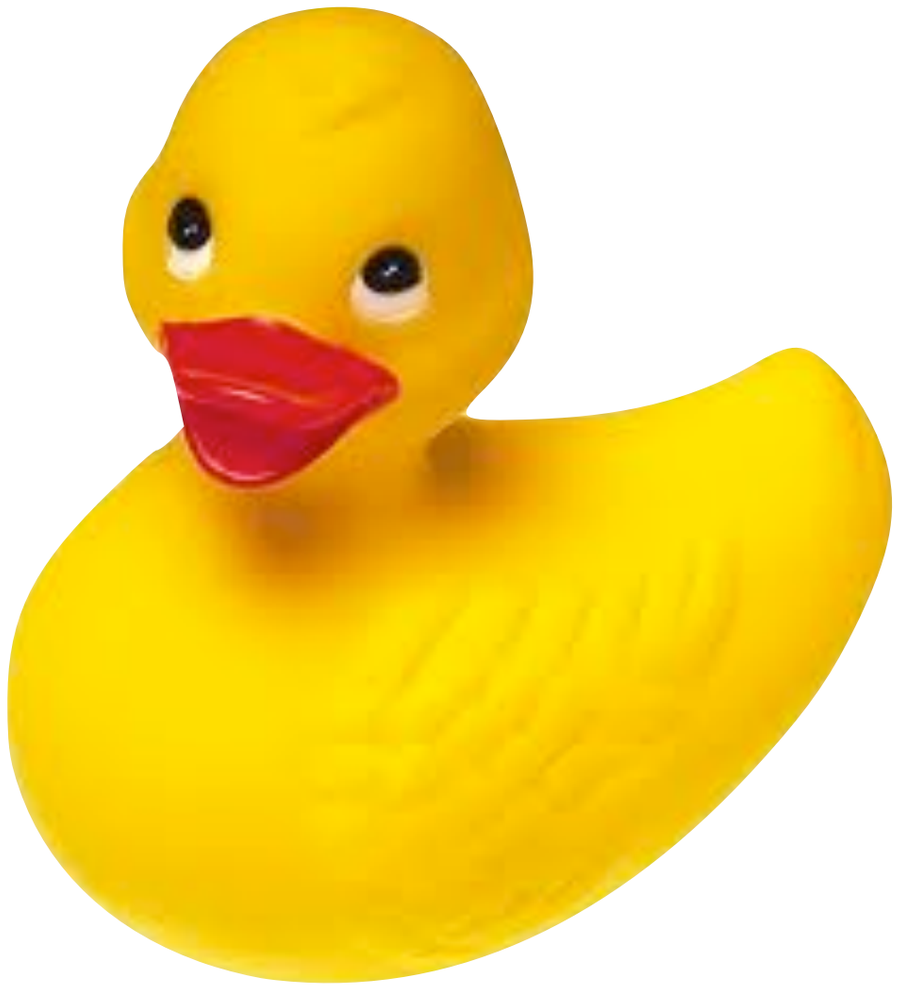
Pueden ser:

```
locales
CONSTANTES
$GLOBALES
@de_instancia
@@de_clase
```

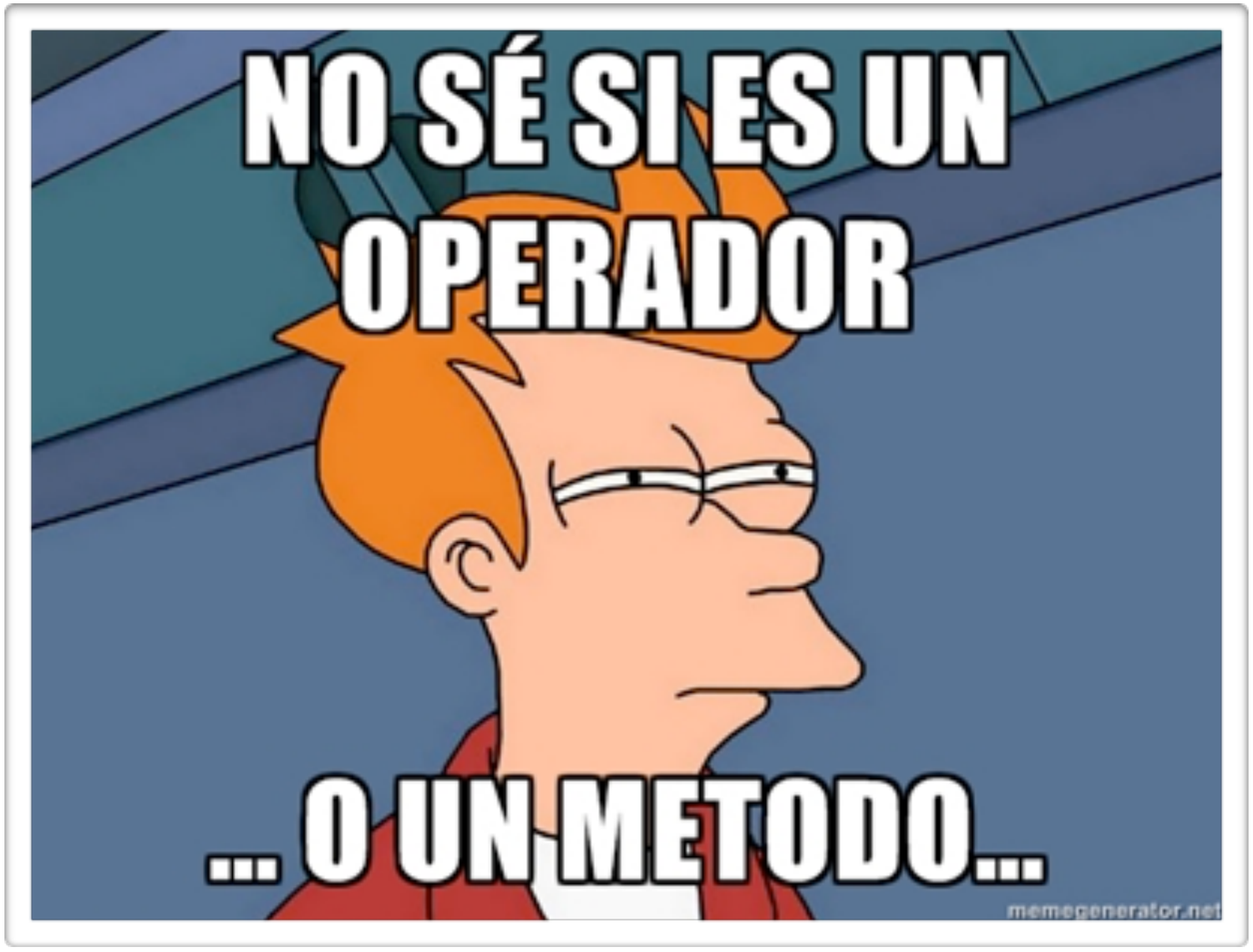
Ruby is **really** Object Oriented

TODO es un objeto

Ruby no tiene interfaces, pero...



Duck Typing



Los operadores son sólo “syntax sugar”... en realidad son métodos...

otros aspectos generales...

- Exception handling
- Garbage collection
- Herencia simple (pero tiene Mixins)
- Bloques, lambdas, procs

Algunos recursos para aprender Ruby

- http://tryruby.org/ (para saborearlo)
- https://www.ruby-lang.org/en/documentation/ (listado de muchos recursos gratuitos)
- Code style guide: https://github.com/bbatsov/ruby-style-guide

VAMOS A LOS DETALLES...

Tipos de Dato

- Text
- Arrays/Hashes
- Symbols
- Objects

Text

- Comillas simples para los literales
  - `'This is a simple Ruby string literal'`
- Comillas dobles permiten interpolación
  - `"360 degrees=#{2*Math::PI} radians"`
  - `"360 degrees=6.28318530717959 radians"`
- Delimitadores arbitrarios
  - `%q(Don't worry about escaping ' characters!)`
  - `%Q|"How are you?", he said|`
  - `%q_This string literal contains \_underscores \__`

“Operadores” de Strings

- + concatenación
- << append
- [] acceso a caracteres

```
planet = "Earth"
"Hello" + " " + planet  # Produces "Hello Earth"

greeting = "Hello"
greeting << " " << "World"
puts greeting     # Outputs "Hello World"
```

```
s = 'hello'
s[0]              # 'h'
s[s.length-1]     # 'o'
s[-1]             # 'o'
s[-2]             # 'l'
s[-s.length]      # 'h'
s[s.length]       # nil
```

Símbolos

- strings inmutables que representan cosas (eficiente)
- símbolos distintos representan contenidos distintos
  - se usa el prefijo :

```
:symbol                         # A Symbol literal
:"symbol"                       # The same literal
:'another long symbol'          # useful for symbols with spaces
s = "string"
sym = :"#{s}"                   # The Symbol :string
```

Arrays

- índices de 0 a size -1
- índices negativos cuentan desde el final
- untyped (elementos pueden ser de distinto tipo)
- tamaño dinámico (se ajustan a la necesidad)

```
[1, 2, 3]           # An array that holds three Fixnum objects
[-10...0, 0..10,]   # An array of two ranges; trailing commas allowed
[[1,2],[3,4],[5]]   # An array of nested arrays
[x+y, x-y, x*y]     # Array elements can be arbitrary expressions
[]                  # The empty array has size 0
words = %w[this is a test]  # Same as: ['this', 'is', 'a', 'test']
open = %w| ( [ { < |        # Same as: ['(', '[', '{', '<']
white = %W(\s \t \r \n)     # Same as: ["\s", "\t", "\r", "\n"]
a = [0, 1, 4, 9, 16]   # Array holds the squares of the indexes
a[0]          # First element is 0
a[-1]         # Last element is 16
a[-2]         # Second to last element is 9
a[a.size-1]   # Another way to query the last element
a[-a.size]    # Another way to query the first element
a[8]          # Querying beyond the end returns nil
a[-8]         # Querying before the start returns nil, too
```

```
a[0] = "zero"    # a is ["zero", 1, 4, 9, 16]
a[-1] = 1..16    # a is ["zero", 1, 4, 9, 1..16]
a[8] = 64        # a is ["zero", 1, 4, 9, 1..16, nil, nil, nil, 64]
a[-9] = 81       # Error: can't assign before the start of an array
a = ('a'..'e').to_a     # Range converted to ['a', 'b', 'c', 'd', 'e']
a[0,0]                  # []: this subarray has zero elements
a[1,1]                  # ['b']: a one-element array
a[-2,2]                 # ['d','e']: the last two elements of the
array
a[0..2]                 # ['a', 'b', 'c']: the first three elements
a[-2..-1]               # ['d','e']: the last two elements of the
array
a[0...-1]            # ['a', 'b', 'c', 'd']: all but the last element
```

Operadores de Array

- + concatenación
- - diferencia (como en conjuntos)
- << append al final del array
- * multiplica un elemento
- | union (elimina duplicados)
- & intersección (elimina duplicados)
- métodos: each, sort, reverse, etc.

Ejemplos

```
a = [1, 2, 3] + [4, 5]  # [1, 2, 3, 4, 5]
a = a + [[6, 7, 8]]   # [1, 2, 3, 4, 5, [6, 7, 8]]
a = a + 9              # Error: righthand side must be an array
a = []          # Start with an empty array
a << 1          # a is [1]
a << 2 << 3     # a is [1, 2, 3]
a << [4,5,6]    # a is [1, 2, 3, [4, 5, 6]]
a = [0] * 8        # [0, 0, 0, 0, 0, 0, 0, 0]
a = [1, 1, 2, 2, 3, 3, 4]
b = [5, 5, 4, 4, 3, 3, 2]
a | b         # [1, 2, 3, 4, 5]: duplicates are removed
b | a         # [5, 4, 3, 2, 1]: same, but order is different
a & b         # [2, 3, 4]
b & a         # [4, 3, 2]
```

Hashes

- también conocidos como maps
- similar a arrays pero mantiene pares key-value (índice es cualquier cosa)

```
numbers = Hash.new     # Create a new, empty, hash object
numbers = {}           # The same
numbers["one"] = 1     # Map the String "one" to the Fixnum 1
numbers["two"] = 2
numbers["three"] = 3
sum = numbers["one"] + numbers["two"]
numbers = { "one" => 1, "two" => 2, "three" => 3 }
numbers = { :one => 1, :two => 2, :three => 3 }  # better
numbers = { one: 1, two: 2, three: 3 }  # the best, new in 1.9
```

Objetos

- todo valor es objeto (no hay tipos primitivos)
- todos los objetos descienden de la clase Object
- siempre se manejan referencias a los objetos (no el objeto mismo)
- si no son literales se crean con new
- garbage collection automático

Ejemplos

```
s = "Ruby"          # Create a String object. Store a reference to it in s.
t = s               # Copy the reference to t. s and t both refer to the
same object.
t[-1] = ""          # Modify the object through the reference in t.
print s             # Access the modified object through s. Prints "Rub".

t = "Java"          # t now refers to a different object.

print s, t          # Prints "RubyJava".
a = "Ruby"          # One reference to one String object
b = c = "Ruby"# Two references to another String object
a.equal?(b)    # false: a and b are different objects
b.equal?(c)    # true: b and c refer to the same object
a == b         # true: but these two distinct objects have equal values
```

Blocks

- fundamentales para métodos iteradores (times, each, map, upto, etc.)
- trozo de código parametrizado anónimo
- separado por { } o do...end
- parámetros al comienzo separados por caracteres | |

Ejemplos

```
3.times {|x| print x }        # => prints “012"
[1,2,3].each {|x| print x } # => prints “123”
(1..3).each  {|x| print x } # => prints "123"
4.upto(6) {|x| print x}      # => prints "456"
squares = [1,2,3].map {|x| x * x} # => [1,4,9]
evens = (1..10).select {|x| x % 2 == 0} # => [2,4,6,8,10]
odds = (1..10).reject {|x| x % 2 == 0}  # => [1,3,5,7,9]
```

Haciendo un Iterador (yield)

- método pasa el control a block mediante instrucción yield

```
def twice
  yield
  yield
end

twice {puts “algo”}
algo
algo

def triple(algo)
  yield algo
  yield algo
  yield algo
end

triple(“debo estudiar Ruby”) {|x| puts x}
debo estudiar Ruby
debo estudiar Ruby
debo estudiar Ruby
```

Otro Ejemplo

```
# This method expects a block. It generates n values of the form
# m*i + c, for i from 0..n-1, and yields them, one at a time,
# to the associated block.

def sequence(n, m, c)
  i = 0
  while(i < n)      # Loop n times
    yield m*i + c   # Invoke the block, and pass a value to it
    i += 1          # Increment i each time
  end
end


sequence(3, 5, 1) {|y| puts y }
1
6
11
```

Métodos

- no hay diferencia entre funciones y métodos
- def nombre(argumentos) ... end
- pueden omitirse paréntesis
- Se puede retornar con return, pero si no siempre se retornará la evaluación de la última sentencia ejecutada

Ejemplos

```
def factorial(n)
  if n < 1                      # Test argument value for validity
    raise "argument must be > 0"
  elsif n == 1                  # If the argument is 1
    1                           # then the value is 1
  else                          # Otherwise, factorial of n is n times
    n * factorial(n-1)     # the factorial of n-1
  end
end

factorial 5  #120
factorial(5) #120
```

```ruby
def prefix(s, len=1)
  s[0,len]
end

prefix("Ruby", 3)      # => "Rub"
prefix("Ruby")         # => "R"
```

```ruby
def max(first, *rest)
   max = first
   rest.each {|x| max = x if x > max }
  max
end

max(1)# => 1 first=1, rest=[]
max(1,2)  # => 2 first=1, rest=[2]
max(1,2,3)   # => 3 first=1, rest=[2,3]
```

Procs y Lambdas

- blocks son anónimos
- procs y lambdas son objetos que representan blocks
- procs se comportan como blocks, lambdas más como métodos
- ambos son instancias de Proc
- son llamados con método call
- lambdas no permiten número variable de argumentos

Ejemplo de proc

```
p = Proc.new {|x,y| x+y }
p.call(2,4)
6
p[2,4]
6
p = Proc.new {|x,y| print x,y }
p.call(1)       # x,y=1:     nil used for missing rvalue:  Prints 1nil
p.call(1,2)     # x,y=1,2:   2 lvalues, 2 rvalues:         Prints 12
p.call(1,2,3)   # x,y=1,2,3: extra rvalue discarded:       Prints 12
p.call([1,2])   # x,y=[1,2]: array automatically unpacked: Prints 12

l = lambda {|x,y| print x,y }
l.call(1,2)     # This works
l.call(1)       # Wrong number of arguments
l.call(1,2,3)   # Wrong number of arguments
l.call([1,2])   # Wrong number of arguments
l.call(*[1,2])  # Works: explicit splat to unpack the array
```

OOP

- no hay una separación nítida entre constructores y otros métodos
- new instancia un objeto y llama a método initialize
- cualquier método de clase puede entonces actuar como constructor

```
class ColoredRectangle
  def initialize(r, g, b, s1, s2)
    @r, @g, @b, @s1, @s2 = r, g, b, s1, s2
  end
  def self.white_rect(s1, s2)
    new(0xff, 0xff, 0xff, s1, s2)
  end
  def self.gray_rect(s1, s2)
    new(0x88, 0x88, 0x88, s1, s2)
  end
  def self.colored_square(r, g, b, s)
    new(r, g, b, s, s)
  end
  def self.red_square(s)
    new(0xff, 0, 0, s, s)
  end
  def inspect
    "#@r #@g #@b #@s1 #@s2"
  end
end

a = ColoredRectangle.new(0x88, 0xaa, 0xff, 20, 30)
b = ColoredRectangle.white_rect(15,25)
c = ColoredRectangle.red_square(40)
```

Atributos

- atributos de instancia comienzan con @

- setters y getters pueden crearse automáticamente (con métodos attr)

Ejemplos

```
class Person
  attr :name, true  # Create @name, name, name=
  attr :age         # Create @age, age
end

class SomeClass
  attr_reader :a1, :a2    # Creates @a1, a1, @a2, a2
  attr_writer :b1, :b2    # Creates @b1, b1=, @b2, b2=
  attr_accessor :c1, :c2  # Creates @c1, c1, c1=, @c2, c2,
c2=
end

p = Person.new
p.name= “John Doe”
puts p.name
```

```
class SoundPlayer

  MAX_SAMPLE = 192

  def self.detect_hardware
    # ...
  end

  def play
    # ...
  end

end
```

```
class SoundPlayer

  MAX_SAMPLE = 192

  def play
    # ...
  end

  def SoundPlayer.detect_hardware
    # ...
  end
```

Atributos y Métodos de Clase

- class variables comienzan con @@
- un ejemplo de class method es new
- class methods van precedidos por nombre de la clase o self
- pueden declararse fuera de la clase

```ruby
class Metal

  @@current_temp = 70
  attr_accessor :atomic_number

  def self.current_temp=(x)
    @@current_temp = x
  end

  def self.current_temp
    @@current_temp
  end

  def liquid?
    @@current_temp >= @melting
  end

  def initialize(atnum, melt)
    @atomic_number = atnum
    @melting = melt
  end

end

aluminum = Metal.new(13, 1236)
copper = Metal.new(29, 1982)
gold = Metal.new(79, 1948)

Metal.current_temp = 1600

puts aluminum.liquid?         # true
puts copper.liquid?           #
false
puts gold.liquid?             #
false

Metal.current_temp = 2100

puts aluminum.liquid?         # true
puts copper.liquid?           # true
puts gold.liquid?             # true
```

Herencia

- Subclase se indica con <
  - class MySubClass < MySuperClass
- No hay herencia múltiple
- Pueden incluirse módulos (mixins)

Ejemplos

```
class Person
  attr_accessor :name, :age, :sex

  def initialize(name, age, sex)
    @name, @age, @sex = name, age, sex
  end

  # ...

end
```

```
class Student < Person
  attr_accessor :idnum, :hours

  def initialize(name, age, sex, idnum, hours)
    super(name, age, sex)
    @idnum = idnum
    @hours = hours
  end

  # ...

end

# Create two objects
a = Person.new("Dave Bowman", 37, "m")
b = Student.new("Franklin Poole", 36, "m", "000-13-5031", 24)
```

Super

```
# This passes a, b, c to the superclass
def initialize(a, b, c, d, e, f)
   super(a, b, c)
end
```

```
# This passes a, b, c to the superclass
def initialize(a, b, c)
   super
end


# This passes no arguments to the superclass
def initialize(a, b, c)
   super()
end
```

Control de Acceso

- private en el cuerpo de una clase hace que métodos que siguen sean privados

- private con parámetros (nombre de métodos) los convierte en privados

- protected methods - objetos de la clase y subclases

```ruby
class Bank
  def open_safe
    # ...
  end

  def close_safe
    # ...
  end

  private :open_safe, :close_safe

  def make_withdrawal(amount)
    if access_allowed
      open_safe
      get_cash(amount)
      close_safe
    end
  end

  # make the rest private

  private

  def get_cash
    # ...
  end

  def access_allowed
    # ...
  end
end
```

Módulos

- módulo provee un espacio de nombres propio
- pueden definirse constantes y métodos al interior de un módulo
- muy útiles para métodos utilitarios
- métodos pueden incluirse en una clase (mixins)
- un módulo puede tener métodos de instancia!
  - en ese caso métodos son parte de la clase que incluya el módulo

Ejemplo mixin

```
module MyMod

  def meth1
    puts "This is method 1"
  end

end
```

```
class MyClass

  include MyMod        # MyMod is mixed into MyClass
  # ...
end


x = MyClass.new
x.meth1                 # This is method 1
```

Ejemplo namespace

```
module Rendering
    class Font  <---------------------------------- Rendering::Font
        attr_accessor :name, :weight, :size
        def initialize( name, weight=:normal, size=10 )
            @name = name
            @weight = weight
            @size = size
        end
        # Rest of the class omitted...
    end
    class PaperSize  <----------------------------- Rendering::PaperSize
        attr_accessor :name, :width, :height
        def initialize( name='US Let', width=8.5, height=11.0 )
            @name = name
            @width = width
            @height = height
        end
        # Rest of the class omitted...
    end
    DEFAULT_FONT = Font.new( 'default' )  <-------- Rendering::DEFAULT_FONT
    DEFAULT_PAPER_SIZE = PaperSize.new
end
```

Módulo Utilitario

```
module WordProcessor
   def self.points_to_inches( points )
      points / 72.0
   end

   def self.inches_to_points( inches )
      inches * 72.0
   end
 # Rest of the module omitted
 end

an_inch_full_of_points =
WordProcessor.inches_to_points( 1.0 )
```

Módulo en Dos archivos

font.rb

```
module Rendering
    class Font
                # Rest of the class omitted...
    end
    DEFAULT_FONT = Font.new( 'default' )
end
```

paper_size.rb

```
module Rendering
    class PaperSize
                # Rest of the class omitted...
    end
    DEFAULT_PAPER_SIZE = PaperSize.new
end
```

```
require 'paper_size'
# full module available
```

Especializando Instancias

- es posible agregar o cambiar el comportamiento de instancias

- es como generar una subclase para ese objeto

Ejemplos

```
a = "hello"
b = "goodbye"

def b.upcase          # create single method
  gsub(/(.)(.)/) { $1.upcase + $2 }
end

puts a.upcase    # HELLO
puts b.upcase    # GoOdBye
```

Expresiones Regulares

- Similares a otros lenguajes
  - Sintaxis: /regexp/
  - Para hacer match: /regexp/ =~ “string”
- caracteres hacen match con caracteres (case sensitive)
  - /123/ =~ “123”, /qwerty1/ =~ “qwerty1”
- . (punto) hace match con un caracter cualquiera
  - /../ =~ “1a”, “34”, “zz”, “[!”
- \ es caracter de escape
  - /3\.14/ =~ “3.14”

- sets de caracteres [aeiou] match con cualquiera del set
- rangos operan como sets [0-2] es [012]
- [0-9a-f][0-9a-f] => hex de dos dígitos
- caracteres especiales \d dígito \w word char \s any space
- | permite alternativa (uno o el otro)
  - A\.M\.|AM|P\.M\.|PM
  - \d\d:\d\d (AM|PM)

- * permite especificar repeticiones
    - A* hace match con cero o mas A´s
    - [aeiou]* hace match con una serie de vocales
- ? hace match con cero o una instancia
- operador de match es =~
    - devuelve nil o el índice del match
- Match posicional
    - ^ - comienzo string $ fin string
        - puts(/^[a-z 0-9]*$/ =~ 'well hello 123') # match
        - puts(/^[a-z 0-9]*$/ =~ 'Well hello 123') # no match

Grupos (Captures)

- se usan paréntesis para especificar match de substrings
- se setean variables $1 a $9 con los matches
  - /(hi).*(h...o)/ =~ "The word 'hi' is short for 'hello'."
  - print($1, " ", $2, "\n") #=> hi hello
  - /(.)(.)(.)/ =~ "abcdef"
  - print($1, " ", $3, "\n") #=> a c

Load y Require

- programa Ruby suele estar repartido en múltiples archivos

- tanto load como require cargan y ejecutan el fuente indicado

- + de un load carga + de una vez archivo

- varios requires cargan solo una vez

Excepciones

```
begin
    # Some code which may cause an exception
rescue <Exception Class>
    # Code to recover from the exception
end
```

```
def calc( val1, val2 )
    begin
        result = val1 / val2
    rescue StandardError => e
        puts( e.class )
        puts( e )
        result = nil
    end
    return result
end
```

```
calc( 20, 0 )
#=> ZeroDivisionError
#=> divided by 0
calc( 20, "100" )
#=> TypeError
#=> String can't be coerced into Fixnum
calc( "100", 100 )
#=> NoMethodError
#=> undefined method `/' for "100":String
```

Excepciones

Se pueden capturar a nivel del bloque del método

```
def calc( val1, val2 )
   val1 / val2
rescue StandardError => e
end
```

StandardError es la excepción base a capturar. Es el default también

```
def calc( val1, val2 )
   val1 / val2
rescue => e # the same as before
end
```

Gems

- RubyGems es el sistema manejador de paquetes y módulos de Ruby
- front end (comando) es gem
- algunos subcomandos útiles incluyen
  - gem list # gems instalados
  - gem install # installar una gem
  - gem update # actualizar una gem o todo
  - gem uninstall # desinstalar