Elementos de Programación en Ruby

Marco Antonio Manzo Bañuelos <amnesiac@unix-power.net>

24-Ene-2004

Resumen

Este tutorial comprende los fundamentos básicos de la programación en el lenguaje Ruby, desde el manejo de los tipos de datos primitivos, hasta una reseña de el trabajo de ruby en problemas del mundo real. Las finalidades son mostrar las bondades que tiene ruby hacia la programación moderna y como este lenguaje se ha ido destacando de entre los demás.

Cabe señalar que ésta no es una guía total para el aprendizaje del lenguaje, existen libros y extensos tutoriales los cuales se pueden consultar en línea o adquiriendolos en las librerías.

El primer capítulo abarca una introducción de ruby, y mención de sus características principales. El capítulo 2 muestra los tipos de datos en ruby y su uso mediante ejemplo sencillos en el intérprete interactivo. El capítulo número 3 comprende el manejo de condiciones y ciclos, asi como una introducción a la construcción de iteradores. El capítulo 4 nos muestra las virtudes de ruby en el uso del paradigma orientado a objetos como tal. Y por último, el capítulo número 6 nos muestra un panorama de como ruby puede ser utilizado para diversas tareas y su capacidad ante ellas.

Índice

1. Breve introducción a Ruby

Ruby es un lenguaje creado por Yukihiro <Matz>Matsumoto `mailto:matz@netlab.jp` bajo la licencia GPL con el fin de proporcionar un lenguaje moderno,

100 % orientado a objetos[1] que sea sencillo de aprender, codificar y mantener, así como presentar una estructura muy elegante.

Ruby hereda muchas funcionalidades y características de lenguajes tales como Perl, Lisp, C++, smalltalk entre otros, con esto quiere decir, que los usos primordiales entran donde estos lenguajes tienen cabe. Es útil para los scripts de tareas cotidianas, procesamiento de texto, tareas de administración de sistemas, en el web, creación de aplicaciones usando interfaces gráficas (tk, gtk), asi como aplicaciones cliente/servidor utilizando sockets.

En el mundo bibliográfico ruby cuenta con extensa documentación en su página principal, así como en otras externas, cuenta con diversos libros desde *cookbooks,* hasta libros de referencia y guias completas (incluso escritas por el propio autor de ruby). [2]

1.1. Características

Entre las principales características de Ruby podemos citar las siguientes:

- Manipulación de excepciones similares a Java y python
- Completo y puramente orientado a objetos: Todo en ruby es un objeto genuino, sin excepción, tal como en smalltalk.
- Es dinámico: tiene la habilidad de agregar clases o instancias aun en tiempo de ejecución.
- Es portable, es desarrollado principalmente en linux pero funciona en: DOS, Windows (cualquiera), MacOS, BeOS, OS/2, etc.
- Cuenta con modulos para diversos tipos de tareas, desde trabajo en web (cgi, xml, rss), bibliotecas para el manejo de bases de datos (mysql/ pgsql), entre otros.
- El motor de expresiones regulares y otras características poderosas para el tratamiento de información es muy al estilo de Perl.
- Contiene un interprete interactivo para hacer funcionar pequeños pedazos de código y hacer pruebas tal como en python. (útil)

1.2. Ruby en la actualidad

Ruby esta en constante desarrollo así como todos aquellos lenguajes donde existe una gran comunidad de desarrolladores, lamentablemente debido a su desarrollo en Japón, es allí donde tomó aun mas fuerza que Perl y Python, mas sin embargo, despues de mas de 8 años de existir, se sigue expandiendo, y haciéndose favorito de muchos desarrolladores en el Occidente.

[1] Es una entidad que contiene datos y control hacia esos datos, la cual va acompañada de atributos y métodos. Es decir es una instancia a una clase.

[2] Un *cookbook* es un libro que plantea diversas problemáticas y nos muestra diversas soluciones, es ideal para ver "Como hago X en Y" y rico en ejemplos.

Actualmente ruby es utilizado en diversas utilerías en sistemas operativos, por ejemplo, en FreeBSD[3] la utilería llamada portupgrade (instalable desde el árbol de ports), esta hecha completamente en ruby, asi como otros scripts de automatización de actividades en dicho SO.

2. Elementos básicos en ruby

2.1. Tipos de datos

En ruby cualquier tipo de dato, función, método, es decir, TODO es un objeto. Por lo tanto ruby presenta los conceptos de encapsulamiento[4], herencia[5], y polimorfismo[6]. Como en los otros lenguajes, una variable válida tiene que comenzar con un caracter (o dash _), pero el caracter debe ser en minúscula y no necesita de algún signo para identificar el tipo de dato o identificador alguno.

Cada uno de estos objetos tienen elementos que los hacen únicos para su tratamiento o uso. Cada objeto contiene un id[7], un tipo y una clase (genralmente estos 2 ultimos campos son el mismo, eso es lo que identifica el tipo de dato del objeto) a la que pertenece el objeto, tal es el caso del siguiente ejemplo:

```
num = 2;
num.class
>>Fixnum
num.id
>>5
num.type
>>Fixnum
```

En ruby, los tipos de datos existentes son los siguientes:

- Fixnum (Entero)
- Float
- String
- Array
- Hash

[3]Proyecto FreeBSD http://www.freebsd.org

[4]Los atributos y métodos de un objeto, están unicamente asociados con ese objeto y nada más.

[5]La herencia permite la extensión de una clase agregando métodos y atributos de una clase padre perteneciente.

[6]Una definición de polimorfismo es el comportamiento que tienen los objetos a responder de diferente (dependiendo la clase) a un mensaje (parametro).

[7]La asignacion de valores a una variable no es mas que la referencia hacia ese objeto, puesto que recordaremos que en ruby los datos primitivos tambien son objetos, por lo tanto es mejor manejarlos de esa manera. Cuando asignamos un entero a una variable, si se copia el valor a dicha variable, puesto que ese objeto no mide mas que un apuntador.

- pseudo-variables (nil (null) / __FILE__ / self)

Y como ejemplo de cada tipo de dato:

```
num = 123 # pertenece a la clase Fixnum, id 247
num = 123.2 #pertenece a la clase Float, id 67267522
string = "ruby world" #pertenece a la clase String, id 67679260
arr = [ "a", "b," ,"c" ] #pertenece a la clase Array, id 67653440
hash = { 'a' =>1, 'b' =>2 } #pertenece a la clase Hash, id 67614160
```

Las constantes en ruby se crean de la misma manera que las variables, es decir, no se necesita algun identificador para indicarle a ruby que se trata de una constante, sin embargo la diferencia es que una constante comienza con mayúscula.

En irb probamos la teoría:

```
>> A = 1
=>1
>> A.type
=>Fixnum
>> A = 2
(irb):20: warning: already initialized constant A
```

2.1.1. Ámbito de las variables

¿Qué es lo que caracteriza entonces si una variable es local, de instancia, global o de clase en ruby?

En ruby, las variables globales, de instancia y de clase, se representan anteponiendo el símbolo correspondiente a cada una:

- `@variable = 3; # Variable de instancia`
- `@@variable = 3; #variable de clase`
- `$variable = 3; # variable global`
- `variable = 3; #variable local ( en su ámbito )`

2.2. Arreglos

2.2.1. Declaración

Un arreglo lo podemos declarar mediante su utilización directa, o llamando el constructor de su respectiva clase.

```
arreglo = [ 1, 2, 3 ]
arreglo = Array.new
arreglo = Array.new(2)
arreglo = []
arreglo = Array.[](1, 2, 3) #lo mismo que la primera
```

Un arreglo puede contener cualquier tipo de dato, inclusive contener otras estructuras anidadas.

```
arr = [ 1, 2, [ "a", "b" ], 3 ]
```

2.2.2. Uso básico

El acceso y modificación a un arreglo es al igual que en los demás lenguajes de programación, por medio de índices.

```
a = [ 1, 3, 5 ]
puts a[0]
>>1
a[0] = 3
b = [ 1, 2, [ "a", "b" ], 3 ]
b[1] = 3
puts b[2][1]
>>"b"
puts b[4]
>>"nil"
puts b
>>"12ab3"
a = [ 1, 2, 3, 4 ]
b = [ "ruby, "perl", "python" ]
b =%w[ ruby perl python ] #equivalente
```

2.2.3. Arreglos como stacks y queues (pilas y colas)

Al igual que en lenguajes como perl y python, los arreglos puden ser utilizandos como los TDA's pila[8] y cola[9] con sus respectivas funciones.

Pilas

```
a = [ 1, 2, 3, 4 ];
puts "Array original: " + a.to_s;
a.pop;
puts "Despues pop: " + a.to_s;
a.push( 1 );
puts "Despues de push: " + a.to_s;
```

El código imprime lo siguiente:

[8]Una pila es un tipo de dato abstracto de tipo LIFO (last in first out), es decir, el ultimo elemento en entrar es el primero en salir. Implementa las funciones de pop/push principalmente.

[9]El TDA cola es de tipo FIFO (first in first out), es decir, el primero en entrar es el primero en salir e implementa principalmente las funciones de shift y unshift (queue/dequeue).

```
Array original: 1234
Despues pop: 123
Despues de push: 1231
```

Colas

```
a = [ 1, 2, 3, 4 ]
puts "Array original: " + a.to_s
a.shift
puts "Despues shift: " + a.to_s
a.unshift( 5 )
puts "Despues de unshift: " + a.to_s
```

El código imprime lo siguiente:

```
Array original: 1234
Despues shift: 234
Despues de unshift: 5234
```

2.2.4. Algunos métodos de la clase Array

Operadores:

- Intersección: `[2, 3, 3, 1 ] & [ 1, 2, 3 ]` *`#Devuelve [1 ,3 ]`*
- Repetición: `[ 1, 3 ] * 2` *`# Devuelve [ 1, 3, 1, 3 ]`*
- Concatenación: `[ 1, 2 ] + [ 4, 5 ]` *`#Devuelve [ 1, 2, 4, 5 ]`*
- Diferencia: `[ 1, 2, 3 ] - [ 1, 3, 5 ]` *`#Devuelve [ 2 ]`*
- Comparación: `[ "a", "b", "c" ] <=> [ "a", "c", "b" ]` *`#Devuelve -1`*
- Unión: `[ "a", "b" ] | [ "b", "c" ]` *`#Devuelve [ "a", "b", "c" ]`*

Métodos:

- assoc:

```
arr1 = [ 1, 2, 3, 4 ]
arr2 = [ 5, 6, 7, 8]
a = [ arr1, arr2 ]
a.assoc( 6 )
>>a = [ 5, 6, 7, 8 ]
```

- map y collect

```
arr = [ "a", "b", "c" ]
arr.collect! { |x| x + 1.to_s }
>>arr = [ "a1", "b1", "c1" ]
```

- delete, delete_at, delete_if

```
arr = [1, 2, 3, 4, 5 ]
arr.delete( 1 )
>>arr = [ 2, 3, 4, 5 ]
arr.delete_at( 2 )
>>arr = [ 2, 3 , 5 ]
arr.delete_if { |x| x <4 }
>>arr = [ 5 ]
```

emtpy?, eql?, include?

```
arr = [ 1, 3, 5, 7, 9 ]
arr.eql?( [ 1, 2, 3 ] )
>>false
arr.empty?
>>false
arr.include?( 7 );
>>true
```

- compact y flatten

```
arr = [ 1, 2, nil, [ 3, 4 ], nil, 5 ]
arr.flatten!
>>arr = [ 1, 2, nil, 3, 4, nil, 5 ]
arr.compact!
>>arr = [ 1, 2, 3, 4, 5 ]
```

- uniq, sort, replace, reverse

```
arr = [ "a", "c", "c", "d", "b", "b" ]
arr.sort
>>arr = [ "a", "b", "b", "c", "c", "d" ]
arr.reverse
>>arr = [ "d", "c", "c", "b", "b", "a" ]
arr.uniq!
>>arr = [ "a", "b", "c", "d" ]
arr.replace( [ 1 ,2 , 3 ,4 ] )
>>arr = [ 1 , 2 , 3, 4 ]
```

2.3. Hashes

2.3.1. Declaración

Un hash (conocido también como arreglo asociativo) es un conjunto de valores, de la forma *key =>value* que tiene numerosas funcionalidades donde el arreglo sencillo no puede entrar.

```
h = { "a" =>1, "b" =>2, "c" =>3 }
```

```
h = Hash.new; #devuelve un hash vacio
h = Hash.new( "valor default" )
h = Hash[ "a" =>1, "b" =>2 ]
```

Al igual que los arreglos, un hash tambien puede contener otro tipo de datos dentro, esto nos permite crear estructuras de datos mas complejas y flexibles para la manipulación de nuestra información. Otras características de los hashes, es que cuando sea crean, no tienen orden alguno, y que las *keys* son únicas.

2.3.2. Uso básico

El acceso a alguno de los valores del hash, se hace mediante su llave.

```
h = { "a" =>3.75, "b" =>4.12, "c" =>[ 2, 3 ] }
puts h["a"]
>>3.75
puts h["c"][1]
>>3
h["a"] = nil
puts h["d"] #Aqui a menos que le mandemos un valor default al crear el hash, nos devuelve nil
>>nil
```

2.3.3. Métodos de la clase Hash

Algunas de los métodos que tienen la clase Array estan implementadas tambien en los hashes, la diferencia pues, es que en hashes trabajaremos con llaves y valores.

- clear, default, default=

```
h = { "a" =>1, "b" =>2, "c" =>3 }
h.default = "default_value"
puts h["f"];
>>"default_value"
puts h.default
>>"default_value"
h.clear
>>h = {}
```

- fetch, delete, delete_if

```
h = { "a" =>5, "b" =>10, "c" =>20 }
h.fetch( "c" )
>>20
h.delete( "b" )
>>h = { "a" =>5, "c" =>20 }
h.delete_if { | k, v | v <10 }
>>h = { "c" =>20 }
```

- emtpy?, has_value?, has_key?, index

```
h = { “a” =>100, “b” =>200, “c” =>300 }
h.empty?
>>false
{}.emtpy?
>>true
h.has_value?( “300” )
>>true
h.has_key?( “f” )
>>false
h.index( “200” )
>>true
```

- sort, invert, update

```
h1 = { “d” =>1, “a” =>2 }
h2 = { “e” =>100, “b” =>200 }
h1.sort
>>[ [ ”a”, 2 ], [ “d”, 1 ] ]
h2.invert
>>h2 = { 100 =>“e”, 200 =>“b” }
h1.update( h2 )
>>h1 = { “d” =>1, “a” =>2, 100 =>“e”, 200 =>“b” }
```

2.4. Cadenas

Una de las partes más importantes de un lenguaje, es el tratamiento de las cadenas, con ello nos permitimos manipular textos de cualquier índole, ya sea con funciones o utilizando de cierta manera expresiones regulares.

En ruby, las cadenas se manejan igual que en los demas lenguajes, y como recordaremos, al igual que los otros tipos de datos, una cadena también es un objeto, por lo tanto una cadena pertenece a una clase (String) la cual contiene métodos para su manipulación.

2.4.1. Métodos básicos para manipular cadenas

La clase String contiene ya métodos predefinidos para el trabajo con cadenas, es decir, muchas de las tareas que realizamos ya existen: conversiones, analizadores, entre otras.

Por ejemplo para separar una cadena delineada por espacios, en componentes léxicos (tokens) aplicamos el método *split*, el cual nos regresa un nuevo arreglo con dichos tokens:

```
cadena = "splitting with split"
cadena.split
>>[ "splitting", "with", "split" ]
"split, split, split".split(", ")
>>[ "split", "split", "split" ]
```

Como lo habiamos mencionado antes, en ruby los tipos de datos son objetos en sí, y los tipos primitivos no son excepción, el ejemplo anterior lo podemos hacer sin la variable cadena de por medio:

```
"splitting with split".split
>>[ "splitting", "with", "split" ]
```

Tal es el caso cuando queremos capitalizar una cadena (es decir convertir el primer caracter a mayúscula) o simplemente hacer transformaciones entre mayusculas y minusculas, esos métodos ya estan implementadas en la clase String:

```
"palabra mayúscula".capitalize
>>"Palabra mayúscula"
"Word".swapcase
>>"wORD"
"word".upcase
>>"WORD"
"WOrd".downcase
>>"word"
```

Los métodos *length y size* nos devuelen el tamaño en bytes de una cadena:

```
"cadena nula".length
>>11
"cadena nula".size
>>11
```

Para añadir una cadena a otra se utiliza el operador *append(<<)*:

```
str = "cadena"
str <<" nueva"
>>"cadena nueva"
```

2.4.2. Subcadenas

En ruby, el acceso y asignación de subcadenas es posible con slices:

```
str = "Esta es una cadena"
newstr = str[13..18]; #newstr = str[13, 6]; o newstr = str[-6, 6]
>>"cadena"
```

O usando expresiones regulares:

```
str = "Esta es otra cadena"
newstr = str[/otra/]
#newstr = str[/o..a/]
#newstr = str[^\w{4}]
>>"otra"
```

Modificando una subcadena usando una expresion regular tambien es sencillo:

```
str = "Ruby es dinámico"
str[/d(.+?)$/] = "poderoso"
>>"Ruby es poderoso"
```

2.5. Expresiones regulares

Las sustituciones, transliteraciones, búsqueda de patrones, etcétera, son procesos clave para la manipulación de información, en ruby estas tareas también son ejecutadas de manera sencilla.

2.5.1. Sustituciones y búsqueda de patrones

Para las substituciones tenemos los siguientes ejemplos:

```
str = "Mas ejemplos con ruby"
str.sub!(/e/, "3")
>>"Mas 3jemplos con ruby"
```

¿Por qué sólo sustutiyó la primer 'e'?

Al igual que en perl, el motor de RE en cuanto encuentra el patron y ya no busca mas (a menos que se le indique lo contrario).

```
str = "Mas ejemplos con ruby"
str.gsub!( /e/, "3" )
>>"Mas 3j3mplos con ruby"
```

Podemos guardar en memoria lo encontrado por un patron dada una RE utilizando los agrupadores ():

```
str = "Ruby perl python"
str.sub(/(\w+) (\w+)/, '\2 \1') #lo que en perl son las variables
$1..$9
>>"perl Ruby python"
```

El método *index*, nos devuelve la posición donde comienza el patrón mandado

```
str = "The Ruby way"
str.index( /way/ )
#str.index("way")
>>10
```

El método *include?* devuelve verdadero si el patrón dado se encuentra en la cadena, falso en caso contrario.

```
str = "The Ruby way"
str.include?("python")
>>false
```

scan devuelve un arreglo con el patron enviado, arreglos anidados en caso de ser varios patrones (al utilizar agrupadores):

```
str = "Ruby regular expressions"
str.scan( /\w+/ )
>>[ "Ruby", "regular", "expressions" ]
```

2.5.2 Transliteracion

La transliteración es efectuada por medio del método *tr*:

```
str = "lame rot13 crypt"
str.tr( "A-Za-z", "N-ZA-Mn-za-m" )
>>"ynzr ebg13 pelcg"
```

(Claro que para encriptar algo tambien ruby cuenta con la función crypt que transforma una cadena al hash DES)

Los ejemplos anteriores todos fueron utilizando RE sobre cadenas (como tal) ahora veremos como utilizar la clase Regexp en ruby.

2.5.2. Clase Regexp

La clase propone métodos utiles para el tratamiento de las expresiones regulares en general. Por ejemplo:

```
re = Regexp.new('\w+')
#re =%r(\w+)
#re = Regexp.compile( .. )
str = "nueva cadena"
str =~ re
>>0
```

Si queremos escapar alguna expresion regular que contenga caracteres que pueden confundirse con nuestro patrón, utilizamos él método *Regexp.escape*(o *Regexp.quote*). El constructor de la clase Regexp, aparte del patron, soporta opciones las cuales son:

`Regexp::EXTENDED` ->ignora espacios vacios o saltos de linea
`Regexp::IGNORECASE` ->no sensitivo al caso
`Regexp::MULTILINE` ->los saltos de línea son tratados como otro caracter

2.6. Funciones

Las funciones en ruby son métodos que no pertenecen a una clase determinada, es decir que no necesitamos de un objeto para llamarlas.

2.6.1. Declaracion

Se declaran por medio de la palabra reservada *def* seguido del nombre de la funcion y opcionalmente el nombre de los parametros esperados:

```
def mifuncion
   puts "Has llamado a mifuncion"
end

mifuncion
```

La funcion imprime:

Has llamado a mifuncion

2.6.2. Parámetros

Los parámetros son declarados en la cabecera de la funcion (después del nombre) de la siguiente manera:

```
def mifuncion2 ( val1, val2 )
   sum = val1 + val2
   puts "La suma de los valores " + val1.to_s + " y " + val2.to_s + " es: " + sum.to_s
end

mifuncion2( 3, 5 )
```

El código imprime:

La suma de los valores 3 y 5 es: 8

3. Ciclos y condiciones

3.1. Condiciones

3.1.1. if/unless

Los condicionales en ruby son los ya conocidos, if y unless

```
if x == 3 then
   sentencia
end

if x != 8 then
   sentencia 1
else
   sentencia 2
end

sentencia if x < 4;
```

Lógica inversa:

```
unless x != 3 then
   sentencia
end

unless x == 8 then
   sentencia1
else
   sentencia2
end

puts "help" unless num.is_a? Fixnum
```

3.1.2. Case

El case en ruby, a diferencia de los demas lenguajes, toma diferentes caminos para evaluar cada uno de los casos, puede evaluar por equivalencia (===) o buscando algun patrón:

```
a = "hello world"

case a
   when "hello"
      puts "Contiene hello"
   when "nada"
      puts "invalida"
   when /\w+/
      puts "re match: " + $~.to_s
   else
      puts "default value"
end
```

En este ejemplo, el case devuelve la tercera comparación, puesto que el patron /\w+/ es verdadero en la cadena evaluada.

3.2. Ciclos

3.2.1. for

Ejemplo de utilización el ciclo for:

```
list = %w[ a b c d e ]

for x in list do
   puts x
end
```

```
for x in 0..list.size do
   puts x
end
```

3.2.2. while/until

```
x = 0;
while x < [ 1, 2, 3, 5 ].size do
   print "#{x} ->"
   x = x.succ
end

x = 0
until x == [1, 2, 3, 4, 5].size do
   puts x
   x = x + 1
end
```

3.3. Iteradores sobre arreglos, hashes y cadenas

La clase Array, String y Hash contienen (y comparten) metodos para la iteracion sobre sus elementos, algunos ejemplos se citan a continuación:

Arreglos

```
[ 1, 2, 3, 4 ].each { | a | print "#{a} + " }
>>1 + 2 + 3 + 4
[ "a", "b", "c"].reverse_each { | x | print "#{x} " }
>>c b a
[ 1, 2, 3, 4 ].each_index { | x | print "#{x} " }
>>0 1 2 3
```

Hashes

```
hash = { "a" =>1, "b" =>2, "c" =>3 }
hash.each { | k, v | print "#{k} ->#{v} --- " }
>>a ->1 --- b ->2 --- c ->3 ---
hash.each_key { | key | puts k }
>>a b c
hash.each_value { | value | puts value }
>>1 2 3
```

Cadenas

```
"Ruby\nworld".each { | x | puts x }
>>Ruby
>>World
```

```
"Ruby world".each(' ') { | a | puts a }
>>Ruby
>>World
"Ruby example".each_byte { | b | print "#{ b } " }
>>82 117 98 121 32 101 120 97 109 112 108 101
```

3.4. Rangos

Existen métodos para iterar por medio de rangos, citamos algunos ejemplos:

```
list = %w[ ab ac ad ae ]

list.size.times do | a |
   puts a * a
end

0.upto( list.size ) do | x |
   print x * x
end

for i in 0..4
   puts i * i
end
```

En el caso del método *times* (que se encuentra en la clase Integer) toma n-1 valores para iterar, mientras que *upto*, toma todo el rango. Estos métodos para iterar sobre rangos serán muy frecuentemente vistos sobre la construcción de iteradores.

3.5. Construcción de iteradores

Muchos de los métodos que sirven para manipular arreglos, hashes, o los métodos para otro tipo de iteraciones (y vaya que muchos) estan basados en una función llamada *yield.* Esta función vista por primera vez parece un poco compleja y engañadora mas sin embargo su propósito principal es ejecutar un bloque de código mandando a ella el valor de la variable a utilizar en el bloque, ejemplos:

```
def iterador
   0.upto(5) do |x|
      yield x
   end
end

iterador { |a| puts "yield yields:" + a.to_s }
```

Aqui podemos observar como yield manda al iterador de la funcion el valor dado y lo ejecuta, tal es el caso de los métodos *collect, map, each,* entre otros, que funcionan con *yield.* Aqui tenemos otro ejemplo con donde construimos nuestro propio metodo collect:

```
def collect2 ( *arr )
   newarr = []
   arr.size.times do |a|
      newarr << yield( arr[a] )
   end
   return newarr
end

newarr = collect2( "a", "b", "c" ) { | x | x + "1" }
puts newarr
```

4. Clases, objetos y métodos en Ruby

4.1. Declaración de clases en Ruby

Las reglas básicas para la construcción de clases en ruby son sencillas:

- Los nombres de clases comienzan con mayúscula
- Los nombres de métodos comienzan con minúscula
- Por default del nombre del constructor es *initialize (al llamar new para crear la instancia)*

Como ya se había mencionado, también es indispensable tener diferentes tipos de variables para el tratamiento de clases, esto es, ruby cuenta con variables de clase y variables de instancia. Las variables de clases son aquellas globales a la clase a la que pertenecen, y pueden ser accesadas desde cualquier lugar en la misma clase o por sus descendientes. Por el otro lado, las variables de instancia son aquellas que guardan datos acerca del objeto (datos de instancia). Veamos un ejemplo:

```
class Books

   @@total = 0

   def initialize( name, author, isbn )
      @name = name
      @author = author
      @isbn = isbn
      @@total = @@total + 1
   end
```

```
   def show
      print " Name: #{ @name }\nAuthor: #{ @author }\nISBN: #{ @isbn }\n"
   end

   def Books.totalbooks
      puts @@total
   end

end

perl = Books.new( "Programming Perl", "Larry Wall", "1-2-234234234" )
perl.show
ruby = Books.new( "Programming Ruby", "Yukihiro Matsumoto", "2-3-1231233" )
ruby.show
Books.totalbooks


El codigo imprime:
Name: Programming Perl
Author: Larry Wall
ISBN: 1-2-234234234
 Name: Programming Ruby
Author: Yukihiro Matsumoto
ISBN: 2-3-1231233
2
```

4.2. Métodos y atributos

Cuando hablamos de métodos hablamos de funciones que pertenecen a una clase, y que van a responder a una instancia de esa clase (es decir un objeto), cuando creamos métodos, muy a menudo requerimos del acceso a variables de instancia (accessors), lo cual podemos escribir de esta manera:

```
#clase Books

   ...

def name

   @name

end

def author

   @author
```

```
end

   ...

perl.name

>>Programming Perl

perl.author

>>Larry Wall
```

Lo que acabamos de crear son *accessors* para lectura, los cuales son muy comúnes, ruby ofrece una manera mas sencilla de utilizar este tipo de *accessors*:

```
#clase Books

...

   attr_reader :name, :author

...

ruby.name

>>Programming Ruby

ruby.author

>>Yukihiro Matsumoto
```

Lo mismo pasa con los *accessors* de escritura, podemos escribir uno de la siguiente manera:

```
#clase Books

...

def name=( new_name )

   @name = name

end

def author=(new_author)

   @author = author

end
```

```
...

ruby.name = "The Ruby Way"

ruby.author = "Hal Fulton"
```

Al igual que para los accessors de lectura, ruby cuenta con una manera de manejarlos mas fácilmente:

```
#clase Books

...

   attr_writer :name, :author

...
```

¿Qué pasa si queremos un accessor para lectura/escritura? tambien ruby nos provee del método para hacerlo:

```
   attr_accessor :var1, :var2
```

4.3. Control de acceso

Como en todos los lenguajes de programación orientados a objetos, siendo el encapsulamiento y abstracción una de las principales características de ellos, contamos también con la capacidad de restringir el acceso a las variables o métodos de una clase:

- ***public***: Este control de acceso permite llamar un método o variable desde cualquier lugar, no hay restricción, por default, los métodos en ruby y las variables son public. (a excepción del método initialize que es privado)
- ***protected:*** El acceso es permitido solo para objetos de la clase definida o subclases.
- ***private:*** Este tipo de acceso solo permite llamar métodos o variables de la clase definida o de los descendientes del mismo objeto.

Eeehhh??? Ok mejor con unos ejemplos:

```
class Usuario

   attr_reader :name, :email

   def initialize ( username, email )
      @name = username
      @email = email
      setpass
```

```
   end

   def setpass
     puts "Password para: #{ @name }"
     @pass = gets.chomp
   end

   def showdata
      print " #{ @name }\n #{ @pass }\n #{ email }\n"
   end

   private :setpass

end

amnesiac = Usuario.new( "Amnesiac", "somewhere@world.com" )
amnesiac.showdata
amnesiac.setpass


En la ejecucion tenemos:
Password para: Amnesiac
asdf
 Amnesiac
 asdf
 somewhere@world.com
class2.ex:26: private method `setpass' called for #<Usuario:0x806e9fc> (NameError)
```

4.4. Herencia y Sobrecarga de operadores

4.4.1. Sobrecarga de operadores

La sobrecarga de operadores es muy sencilla en ruby, basta con utilizar el método *alias:*

```
class Fixnum

   alias suma +
   alias resta -

def +(num2)

   ...

end

end
```

```
...

total = 1 + 2

total = 1.suma(2)
```

4.4.2. Herencia

La herencia en ruby es manejada con el operador $<$ dado el nombre de la clase padre:

```
#class Books
   ...
#end


class RubyBooks < Books

   @@rubybooks = 0

   def initialize( name, author, isbn, press )
      super( name, author, isbn )
      @press = press
      @@rubybooks += 1
   end

   def rubybooks
      puts @@rubybooks
   end

end

book = Books.new( "Object Oriented Perl", "Damian Conway", "3-2-11231231231231" )
r = RubyBooks.new( "Ruby in a nutshell", "Yukihiro Matsumoto", "1-2-23423423432", "O'Reilly'
r.rubybooks
Books.totalbooks
r.show


El codigo muestra lo siguiente:
1
2
 Ruby in a nutshell
Yukihiro Matsumoto
1-2-23423423432
```

4.5. Modulos, Mixins y Singletons

4.5.1. Modulos

Los modulo nos ayudan a sintetizar el trabajo organizando en pequeños bloques de construcción nuestro codigo, ademas de proveer la interfaz básica de creación de los *mixins*.

Un modulo es declarado con la palabra clave *module* al inicio del archivo, y de allí en delante hacemos la definicion de nuestro modulo:

```
module HTMLParser
... #aqui va la declaración de las funciones/clases del modulo
end
```

y en el archivo donde queremos cargarlo hacemos:

```
require "HTMLParser"
...
```

4.5.2. Mixins

¿Qué pasa con la herencia múltiple en ruby ?

Bueno, como es sabido la herencia múltiple puede ser muy ambigua y puede provocar confusiones, pues bien ruby se maneja de una manera muy peculiar, por medio de los *mixins,* éstos son módulos que contienen declaraciones de diferentes clases, las cuales van a heredar a una clase hija, por ejemplo:

```
module Mixin

   class Clase1
   ...
   end
   class Clase2
   ...
   end

...

a = Clase1.new

b = Clase2.new

a.class2method

b.class1method
```

4.5.3. Singletons

Los singletons en ruby se presentan en 2 sabores, lo que es el *diseño de patrones* y lo que son *singleton methods*. En el primer caso, los singletons en el diseño de patrones, son aquellos donde una clase solo puede tener una instancia, por ejemplo:

```
class Administrator
   private_class_method :new

   @@admin = nil

   def Admin.newadmin
      @@admin = new unless @@admin
      @@admin
   end

end
```

En este caso al crear un nuevo objeto con nuestro nuevo constructor, vamos a ver que cada que creemos uno, el objeto va a seguir referenciando al mismo id del primero, es decir, siempre tendremos la misma instancia.

Los singleton methods son aquellos los cuales solo responden al llamado de una misma instancia, es decir, a la instancia a la que pertenecen.

```
def object.singleton( name )

   @name = name
   puts @name

end

...

object.singleton( "parametro" )
```

5. Ruby y lo demás...

5.1. Ruby y el manejo de estructuras de datos (Archivos, XML, YAML)

- La clase de File cuenta con los métodos necesarios para trabajar con archivos, creación, modificación y manipulación de cualquier tipo de archivo
- Cuenta con parsers y creadores de archivos XML válidos, asi como el manejo de RSS y XSLT

- La biblioteca YAML cuenta con opciones de parsing, creacion y modificación de documentos YAML
- Bibliotecas para el trabajo con archivos HTML.

Ejemplo:

```
file = File.new( "httpd-access.log", "a" )
file.puts "[12:34:00] Hey! Alguien nos ataca!! ah no... mentira"
file.close
```

5.2. Ruby en el web

- mod_ruby
- eRuby
- CGI
- FastCGI

5.3. Ruby y las bases de datos

- DBM
- MySQL
- PostgreSQL
- otras

5.4. IPC en Ruby

- Sockets
- Threads
- Pipes
- Distributed Ruby (drb)

5.5. Ruby y las interfaces gráficas

- Ruby/Tk
- Ruby/Qt
- Ruby/Gnome2[, Ruby/Gnome2]
- Ruby/Delphi (Apollo)

6. Conclusiones

Y... la mejor forma de ver que tan poderoso es ruby, es utilizándolo. ;)

Referencias

[1] Thomas David, Hunt Andrew, *Programming Ruby: A pragmatic programmer's guide,* Addison-Wesley, 1ra ed, Diciembre 2000, ISBN: 0201710897

[2] *Programming Ruby, A pragmatic programmer's guide Online,* Programming Ruby Online `http://www.rubycentral.com/book`

[3] Fulton Hal, *The Ruby Way*, SAMS, 1ra ed, Diciembre 2000, ISBN: 0672320835

[4] *Introduction to Ruby,* Daniel Carrera, Introduction to Ruby `http://www.math.umd.edu/%7Edcarrera/ruby/0.3/`

[5] *Ruby Homepage*, Ruby Homepage `http://www.ruby-lang.org`

[6] The Ruby Central, The Ruby Central `http://www.rubycentral.com`

[7] Ruby/Gnome2 Tutorial, Ruby/Gnome2 Tutorial `http://ruby-gnome2.sourceforge.jp/hiki.cgi?tutorials`