Programación en Ruby

Programación en Ruby

Introducción

Ruby es un lenguaje de programación interpretado, reflexivo y orientado a objetos, creado por el programador japonés Yukihiro "Matz" Matsumoto, quien comenzó a trabajar en Ruby en 1993, y lo presentó públicamente en 1995. Combina una sintaxis inspirada en Python, Perl con características de programación orientada a objetos similares a Smalltalk. Comparte también funcionalidad con otros lenguajes de programación como Lisp, Lua, Dylan y CLU. Ruby es un lenguaje de programación interpretado en una sola pasada y su implementación oficial es distribuida bajo una licencia de software libre.

Historia

El lenguaje fue creado por Yukihiro "Matz" Matsumoto, quien empezó a trabajar en Ruby el 24 de febrero de 1993, y lo presentó al público en el año 1995. En el círculo de amigos de Matsumoto se le puso el nombre de "Ruby" (en español rubí) como broma aludiendo al lenguaje de programación "Perl" (perla). La última versión estable es la 1.8.6, publicada en diciembre de 2007. El 26 de ese mismo mes salió Ruby 1.9.0, una versión en desarrollo que incorpora mejoras sustanciales en el rendimiento del lenguaje, que se espera queden reflejadas en la próxima versión estable de producción del lenguaje, Ruby 1.9.0.1 Diferencias en rendimiento entre la actual implementación de Ruby (1.8.6) y otros lenguajes de programación más arraigados han llevado al desarrollo de varias máquinas virtuales para Ruby. Entre ésas se encuentra JRuby, un intento de llevar Ruby a la plataforma Java, y Rubinius, un intérprete modelado basado en las máquinas virtuales de Smalltalk. Los principales desarrolladores han apoyado la máquina virtual proporcionada por el proyecto YARV, que se fusionó en el árbol de código fuente de Ruby el 31 de diciembre de 2006, y se dará a conocer como Ruby 1.9.0.1.

Tutorial de Instalación

Descargando Ruby

Como lenguaje multiplataforma, Ruby ha sido portado a distintos sistemas operativos y arquitecturas. Esto significa que si tu desarrollas un programa en un PC (por ejemplo), será posible ejecutarlo en otra máquina distinta como es un MAC (por poner otro ejemplo).

Las siguientes instrucciones son para instalar Ruby en una PC. Para otras plataformas, ver este capítulo [1] de la Why's (poignante) guide to Ruby.

La forma más sencilla de instalar Ruby en un PC es mediante el **Ruby One-Click Installer**. Después de descargarlo instálalo aceptando todo por defecto. Al instalarse, las Variables de Entorno del Sistema son actualizadas, de tal forma que se incluya el directorio del ejecutable de Ruby: gracias a esto, podrás ejecutarle desde cualquier drectorio en tu PC.

La instalación de Ruby incluye la primera edición del libro "Programming Ruby" y el editor SciTe.

Los directorios de la instalación

Supongamos que la instalación de Ruby fue en **c:/ruby**.

Esta instalación creó una serie de directorios:

c:/ruby/bin es donde los ejecutables son instalados (incluyendo ruby e irb).

c:/ruby/lib/ruby/1.8 aquí están programas escritos en ruby. Estos ficheros son librerías de Ruby: proveen funcionalidades que puedes incorporar en tus programas.

c:/ruby/lib/ruby/1.8/i386-mswin32 contiene una serie de extensiones específicas del PC. Los ficheros en esta terminación acaban en .so o .dll (dependiendo de la plataforma). Estos ficheros con extensiones programadas en lenguaje C; dicho de otra forma: son fihceros binarios, compilados durante el proceso de instalación, que se pueden ejecutar desde Ruby.

c:/ruby/lib/ruby/site_ruby aquí es donde el administrador de tu sistema y/o tú podéis almacenar las extensiones y librerías de terceras partes: código escrito por ti mismo, o por otros.

c:/ruby/lib/ruby/gems es el sistema Ruby-Gemes, encargado de la instalación de nuevas herramientas.

c:/ruby/src es donde se halla el código fuente de Ruby.

c:/ruby/samples/RubySrc-1.8.6/sample aquí podrás encontrar algunos programas ejemplo escritos en Ruby.

Programa "Hola Mundo!"

Arranca el IRB (Interactive Ruby).

La pantalla nos muestra:

```
irb(main):001:0>
```

Escribe esto: "Hola Mundo" y luego ENTER

La pantalla muestra:

```
irb(main):001:0> "Hola Mundo"
=> "Hola Mundo"
```

¡Ruby te obedeció!

¿Qué fue lo que pasó? ¿Acaso acabamos de escribir el programa “Hola Mundo” más corto del mundo? No exactamente. La segunda linea sólo es la forma que tiene IRB para decirnos el resultado de la última expresión evaluada. Si queremos que el programa escriba “Hola Mundo” necesitamos un poco más:

```
irb(main):002:0> puts "Hola Mundo"
Hola Mundo
=> nil
```

puts es el comando básico para escribir algo en Ruby. Pero entonces, ¿qué es ese => *nil*? Ese es el resultado de la expresión *puts* siempre retorna *nil*, que es el valor que significa “absolutamente nada” en Ruby.

Diccionario de Palabras Reservadas, Operadores y Símbolos del Lenguaje

Palabra Reservada	Función
alias	Crea un alias para un operador, método o variable global que ya exista.
and	Operador lógico, igual a && pero con menor precedencia.
break	Finaliza un *while* o un *until loop*, o un método dentro de un bloque
case	Compara una expresión con una clausula *when* correspondiente
class	Define una clase; se cierra con *end*.
def	Inicia la definición de un método; se cierra con *end*.
defined?	Determina si un método, una variable o un bloque existe.
do	Comienza un bloque; se cierra con *end*.
else	Ejecuta el código que continua si la condición previa no es *true*. Funciona con *if*, *elsif*, *unless* o case.
elsif	Ejecuta el código que continua si la condicional previa no es *true*. Funciona con *if* o *elsif*.
end	Finaliza un bloque de código.
ensure	Ejecuta la terminación de un bloque. Se usa detrás del ultimo *rescue*.
false	Lógico o Booleano *false*.
true	Lógico o Booleano *true*.
for	Comienza un loop *for*. Se usa con *in*.
if	Ejecuta un bloque de código si la declaración condicional es *true*. Se cierra con *end*.
in	Usado con el loop *for*.
module	Define un modulo. Se cierra con *end*.
next	Salta al punto inmediatamente después de la evaluación del loop condicional
nil	Vacio, no inicializado, invalido. No es igual a cero.
not	Operador lógico, igual como !.
or	Operador lógico, igual a // pero con menor precedencia.
redo	Salta después de un loop condicional.
rescue	Evalua una expresión después de una excepción es alzada. Usada después de *ensure*.
retry	Cuando es llamada fuera de *rescue*, repite una llamada a método. Dentro de *rescue* salta a un bloque superior.
return	Regresa un valor de un método o un bloque.
self	Objeto contemporáneo. Alude al objeto mismo.
super	Llamada a método del mismo nombre en la superclase.
then	Separador usado con *if*, *unless*, *when*, *case*, y *rescue*.
undef	Crea un método indefinido en la clase contemporánea.
unless	Ejecuta un bloque de código si la declaración condicional es *false*.
until	Ejecuta un bloque de código mientras la declaración condicional es *false*.
when	Inicia una clausula debajo de *under*.
while	Ejecuta un bloque de código mientras la declaración condicional es *true*.
yield	Ejecuta un bloque pasado a un método.
FILE	Nombre del archivo de origen contemporáneo.
LINE	Numero de la linea contemporánea en el archivo de origen contemporáneo.

Tipos de Datos Fundamentales disponibles en el Lenguaje

Datos Fundamentales

Números

En ruby, todo es tratado como un objeto, eso no excluye a los números, en forma general, ruby cuenta con diferentes clases para manejar cada tipo de números, por ejemplo:

Integer -> la clase base de donde derivan todos los enteros.

Fixnum -> clase para números enteros, su tamaño depende de la arquitectura de donde se interprete el código, sin embargo, su tamaño es eso -1 bit y usa complemento 2 para su representación en memoria, si un número excede el tamaño asignado, automaticamente se convierte en bignum.

Bignum -> Contiene valores mayores a fixnum, la restricción depende de la arquitectura pero pueden guardarse números muy grandes, tanto como de nuestra memoria, si el número ingresado cabe en un fixnum, automáticamente se convierte a esta clase. Float -> Almacena números con punto flotante con la arquitectura de doble precisión nativa.

Rational -> Almacena números racionales, es decir, números con un valor de numerador y un denominador.

Strings

Para las cadenas de caracteres, su uso es bastante similar al de cualquier lenguaje orientado a objetos, con la clase String, sin embargo cabe mencionar algunas caracteristicas mas emocionantes de ruby, por ejemplo, puedes multiplicar cadenas!

“Hola ” * 5 => “Hola Hola Hola Hola Hola”

Para una vista mas completa referirse al capitulo de Strings

Date/Time

Ruby ya trae librerias para manejar los tipos date y time, sin embargo para usarlos, debemos llamar a la librería correspondiente, esto con la palabra reservada require

require ‘date’

Con esto ya podemos usar el tipo Date, este contiene un fecha(dia, mes, año), sin embargo, si deseas agregar la hora debemos hacer uso de otra clase: DateTime, que contiene, ademas de la fecha, la hora, minutos y segundos de la instancia que creemos.

Operaciones de Entrada y Salida

Operaciones Básicas de I/O

El método básico para imprimir algo en Ruby es usando print(), o, si queremos espaciado println(), como en c++

print(“Hola mundo”)

=> Hola Mundo

Recordemos que ruby es un lenguaje interpretado, por lo que no es necesario todo un gran protocolo para imprimir algún valor, genial no?

Ademas, hay otra forma de mandar una salida, miren

puts “Hola mundo, de nuevo” => Hola mundo, de nuevo

Todo esta bien hasta aca, pero… ¿Y si queremos recibir alguna entrada?, eso tambien es simple, usamos el comando gets, veamos

```
puts "¿Como te llamas?" name = gets.chomp
puts "Hola " + name + "!"
```

eso te pide te nombre, lo guarda en la variable “name” y lo utiliza para imprimir un saludo.

Tipos de datos Complejos

Representación de una clase

Una clase en **Ruby** se representa por la palabra reservada class seguida del nombre de la clase. El nombre de la misma se comienza normalmente con una letra mayúscula. La declaración de la clase se cierra con la palabra reservada end.

Ejemplo:

```
class Lenguaje
   #esta clase cuenta con un constructor sin parámetros
      def initialize
      …
      …
      end
   end
```

Para utilizar esta clase se declara una variable y se inicializa de la siguiente manera:

miLenguaje = Lenguaje.new

Declaración de un arreglo

Un arreglo es una colección de elementos que pueden ser accesados por un índice. En este caso el índice es un valor entero y los mismos comienzan desde cero. Se utiliza el nombre de la variable junto con brackets para referirse a un valor específico.

Ejemplo:

```
   numeros = [ 3, 4, 12, 22.5 ]
   colores = [ “rojo”, “azul”, “verde” ]
   #sacando el primer elemento de numeros y el segundo de colores
   puts(numeros[0])
   puts(colores[1])
```

Declaración e inicialización de Variables

Declaración e inicialización de variables

En **Ruby** no hay una declaración explícita del tipo de dato de una variable. El mismo se determina en tiempo de ejecución a partir del valor que se le ha asignado. Esto significa que las variables pueden tomar diferentes tipos de datos en diferentes partes del programa. Para determinar si un nombre hace referencia a una variable o a un método, Ruby utiliza heurística, en la cual a medida que va leyendo el código fuente va llevando control de los símbolos que ha leído, asumiendo que los mismos son variables. Cuando se encuentra con un símbolo que puede ser una variable o un método, revisa si ya se le ha asignado algo anteriormente. Si es así, entonces lo trata como variable; si no, lo trata como un método.

Ejemplo:

```
   #declaración de una variable de tipo String
   Nombreclase = “Lenguajes de Programación”
   #declaración de una variable de tipo entero
   Numeroalumnos = 30
   #declaración de un arreglo
```

```
numeros = [ 3, 4, 12, 22.5 ]
```

Tabla de Operadores con Asociatividad y Precedencia

Tabla de operadores con precedencia de más alta a más baja

Método*	Operador	Descripción	
SI	[ ] [ ]=		
SI	**	Exponente	
SI	! ~ + -	Not, complemento, más y menos unarios	
SI	* / %	Multiplicación, división, módulo	
+ -	Más, menos		
SI	>> <<	Shift a la derecha e izquierda	
SI	&	Bitwise And	
SI		Bitwise Or y Or regular	
SI	<= < > >=	Operadores de comparación	
SI	<=> == === != =~ !~	Operadores de igualdad y coincidencia de patrones	
	&&	And lógico	
	ll	Or lógico	
	.. ...	Rango incluyente y excluyente	
	? :	If-then-else ternario	
	= &= >>= <<= *= &&=	= **=	Asignación
	defined?	Revisar si un símbolo está definido	
	not	Negación lógica	
	or and	Composición lógica	
	If unless while until	Modificadores de expresión	
	begin end	Expresiones de bloque	

```
*Operadores marcados con “SI” pueden tratarse como métodos, y por lo tanto se pueden sobreescribir.
```

Ruby soporta hacer corto circuito en los operadores de comparación.

Estructuras de Decisión e Iteración

Estructuras Simples

En Ruby, sólo nil y false son evaluados como falso, todo lo demás (incluyendo 0), es verdadero. En Ruby nil es un objeto. Puedes agregar métodos a nil como a cualquier otro objeto y puedes llamar los métodos de nil como los de cualquier otro objeto. Vamos a explorar algunas estructuras muy simples disponibles en Ruby. El ejemplo que sigue (p014estructuras.rb) muestra la estructura if else end. Es costumbre en ruby no poner paréntesis con if y while.

```
1 #En Ruby, ninl y flase son evaluados como falso
2 #todo lo demas (incluyendo 0) es verdadero.
3 # nil es un objeto
4 # if else end
5 var = 5
```

```
6 if var > 4
7   puts "La variable es mayor que 4"
8   puts "Puedo tener muchas declaraciones aqui"
9   if var == 5
10     puts "Es posible tener if y else anidados"
11   else
12     puts "Too cool"
13   end
14 else
15   puts "La variable no es mayor que 4"
16   puts "Puedo tener muchas declaraciones aqui"
17 end
18
19 # Loops
20 var = 0
21 while var < 10
22   puts var.to_s
23   var += 1
24 end
```

Un ejemplo del uso de elsif está en el programa p015elsif.rb :

```
1 # ejemplo del uso de elseif
2
3 # Ejemplo original
4 puts "Hola, cual es tu nombre?"
5 STDOUT.flush
6 nombre = gets.chomp
7 puts 'Hola, ' + nombre + '.'
8
9 if nombre == 'Juan'
10         puts 'Que bonito nombre!!'
11 else
12         if nombre == 'Ivan'
13                 puts 'Ivan es Juan en ruso!!'
14         end
15 end
16
17 # Ejemplo modificado usando elsif
18 puts "Hola, cual es tu nombre?"
19 STDOUT.flush
20 nombre = gets.chomp
21 puts 'Hola, ' + nombre + '.'
22
23 if nombre == 'Juan'
24         puts 'Que bonito nombre!!'
25 elsif nombre == 'Ivan'
26         puts 'Ivan es Juan en ruso!!'
```

```
27 end
28
29 # Otra modificacion
30 puts "Hola, cual es tu nombre?"
31 STDOUT.flush
32 nombre = gets.chomp
33 puts 'Hola, ' + nombre + '.'
34
35 # || es el operador logico 'o'
36 if nombre == 'Juan' || nombre == 'Ivan'
37         puts 'Que bonito nombre!!'
38 end
```

Algunos operadores condicionales comunes son: ==, != <=, >=, <, > Ruby tiene también una forma negativa de if. La estructura unless comienza con la palabra unless y termina con end. El cuerpo es el texto que aparece en medio de las dos. A menos que la expresión que continúe a la palabra unless sea verdadera, el cuerpo es ejecutado, de lo contrario el intérprete lo ignora.

```
1 unless ARGV.length == 2
2   puts "Usage: program.rb 23 45"
3   exit
4 end
```

En el programa anterior, el cuerpo es ejecutado a menos que el número de elementos en el array sea igual a 2. El método Kernel.exit termina el programa, regresando un valor de status al sistema operativo. Loops como while están disponibles. Observa el siguiente ejemplo:

```
1 #Loops
2
3 var = 0
4 while var < 10
5   puts var.to_s
6   var += 1
7 end
```

1.Escribe un programa (p016bisiesto.rb) que pregunte al usuario un año cualquiera y determine si es bisiesto o no. 2.Escribe un método llamado bisiesto que acepte un año, determine si es bisiesto y calcule el número total de minutos para ese año. (p017metodobisiesto)

Para determinar si un año es bisiesto: 1.Si el año es divisible ente 4, ve al paso 2. Si no, ve al paso 5. 2.Si el año es divisible ente 100, ve al paso 4. Si no, ve al paso 5. 3.Si el año es divisible ente 400, ve al paso 4. Si no, ve al paso 5. 4.El año es bisiesto. (Tiene 365 días). 5.El año no es bisiesto. (Tiene 365 días).

Expresiones case Esta estructura funciona de manera muy similar a una serie de expresiones if: te permite enumerar una serie de condiciones y ejecutar una expresión que corresponda al primer valor que sea verdadero. Por ejemplo, los años bisiestos deben ser divisibles entre 400 o entre 4 y no entre 100. Ten presente que case regresa el valor de la última expresión ejecutada.

```
1 year = 2000
2
3 leap = case
4        when year % 400 == 0: true
```

```
5         when year % 100 == 0: false
6         else year % 4 == 0
7         end
8
9 puts leap
10
11 #el resultado es: true
```

Declaración, definición y uso de Métodos y Funciones

Métodos

los métodos en ruby se definen utilizando la palabra reservada def al iniciar la rutina y end al finalizarla

```
def Metodo
end
```

para mandarle parametros, solo debemos escribir entre parentesis estos, separados por comas

```
def Suma(Operando1, OPerando2)
      puts Operando1 + Operando2
end
```

para especificar que el método retornará algun valor, solo debemos usar la palabra reservada return

```
def Suma(Operando1, OPerando2)
      sum = Operando1 + Operando2
      return sum
end
```

sin embargo, es importante hacer notar que de todas maneras ruby retorna la ultima expresión evaluada en el método, por lo tanto escribir

```
def met
      return "Hola"
end
```

es igual a

```
def met
      "Hola"
end
```

Implementación y uso de la Programación Orientada a Objetos

Escribiendo nuestras propias clases

Hasta ahora hemos usado el estilo de programación procedural (que todavía se utiliza en lenguajes como C) para escribir nuestros programas. Programar de manera procedural significa que nos enfocamos en los pasos necesarios para completar la tarea sin prestar atención a cómo son manipulados los datos.

En el estilo de programación Orientado a objetos, los objetos son tus agentes, tus proxies, en el universo de tu programa. Tu les pides información, les asignas tareas, les pides que realicen cálculos y te los reporten y los comunicas entre si para que trabajen juntos. Cuando diseñes una clase, piensa en los objetos que van a ser creados de ella. Piensa acerca de lo que los objetos conocen y las cosas que hacen.

Las cosas que un objeto sabe de si mismo se llaman variables de instancia (instance variables). Representan el estado de un objeto (los datos, por ejemplo, la cantidad y el id de un producto) y pueden existir valores únicos para cada instancia de la clase. Las cosas que un objeto puede hacer se llaman métodos. Un objeto es una entidad que sirve como contenedor de datos y también controla el acceso a los mismos. Asociados con el objeto, hay una serie de atributos que esencialmente no son más que variables que le pertenecen al objeto. También asociadas con el objeto, hay una serie de funciones que proveen una interface para la funcionalidad del objeto, llamadas métodos. Hal Fulton Una clase es una combinación de estado y métodos y es utilizada para construir objetos. Es una especie de plano para un objeto. Por ejemplo, es posible que utilices una clase llamada Boton para hacer decenas de botones diferentes y cada uno puede tener su propio color, tamaño, forma, etiqueta, etc. Un objeto es una instancia de una clase.

Lee esto con atención!

Las clases en ruby son objetos de primera clase, cada una es una instancia de la clase Class. Cuando una nueva clase es definida (de manera típica usando class Nombre end), un objeto de clase Class es creado y asignado a una constante (Nombre en este caso). Cuando llamamos Nombre.new para crear un objeto, el método de instancia en la clase Class es llamado, que a su vez llama al método allocate que asigna memoria para el objeto antes de finalmente llamar el método initialize. Las fases de construcción e inicialización de un objeto son independientes y ambas pueden ser sobreescritas. La inicializacón se hace mediante el método de instancia initialize mientras que la construcción se hace vía el método de clase new. initialize no es un constructor!. La siguiente Jerarquía de clases es informativa. Vamos a escribir nuestra primera clase simple. p029perro.rb

```
1 # definicion de la clase Perro
2 class Perro
3   def initialize(raza, nombre)
4     # Variables de instancia
5     @raza = raza
6     @nombre = nombre
7  end
8
9   def ladra
10     puts 'Ruff! Ruff!'
11  end
12
13   def muestra
14     puts "Soy de raza #{@raza} y mi nombre es #{@nombre}"
15   end
16 end
17
18 # Crear un objeto
```

```
19 # Los objetos son creados en el 'heap'
20 p = Perro.new('Labrador', 'Benzy')
21
22 =begin
23   Cada objeto 'nace' con ciertas abilidades innatas.
24   Para ver una lista de los métodos con los que nace un objeto
25   puedes llamar el metodo methods
26 =end
27 puts p.methods.sort
28
29 # Entre todos estos metodos, oject_id y respond_to? son
30 # importantes. Cada objeto en ruby tiene asociado un numero
31 # unico.
32 puts "El id del objeto es #{p.object_id}."
33
34 # Para saber si un objeto sabe como responder a un mensaje
35 # puedes usar el metodo respond_to?
36 if p.respond_to?('habla')
37   p.habla
38 else
39   puts 'Lo siento, el objeto no entiende el mensaje habla.'
40 end
41
42 p.ladra
43 p.muestra
44
45 # hacer que d y d1 hagan referencia al mismo objeto
46 p1 = p
47 p1.muestra
48
49 # hacer que d haga referencia a nil
50 p = nil
51
52
53 # Si ahora declaramos:
54 p1 = nil
55 # entonces el Objeto de clase perro es abandonado y es
56 # elegible para la recoleccion de basura (GC)
```

Si ejecutamos el programa, además de la lista de métodos con los que nace el objeto 1, obtenemos como resultado:

```
El id del objeto es 281140.
Lo siento el objeto no entiende el mensaje habla.
Ruff! Ruff!
Soy de la Raza Labrador y mi nombre es Benzy
Soy de la Raza Labrador y mi nombre es Benzy
```

El método new es usado para crear un objeto de clase Perro. Los objetos son creados en el heap. La variable p es conocida como una variable de referencia. No guarda el objeto en si, sino algo parecido a un apuntador o una

dirección del objeto. Utilizamos el operador punto (.) en una variable de referencia para decir, "utiliza lo que está antes del punto para traerme lo que está después del punto". Por ejemplo: p.ladra.

Si estás escribiendo una aplicación Rails en la que uno de tus modelos es, digamos, Cliente, entonces cuando escribes el código que hace que las cosas pasen (un cliente accesando un sitio, actualizando un número de teléfono de un cliente, agregando un artículo a un carrito de compras), con toda seguridad estarás enviando mensajes a objetos de la clase Cliente. Aún recién creado, un objeto no está completamente en blanco. Tan pronto como un objeto comienza a existir, responde a algunos mensajes. Cada objeto 'nace' con ciertas habilidades innatas. Para ver una lista de los métodos con los que nace un objeto, puedes usar el método methods.

Es posible determinar de antemano (antes de pedirle a un objeto que haga algo) si un objeto es capaz de manejar determinado mensaje usando el método respond_to?. Este método existe para todos los objetos; puedes preguntarle a cualquier objeto si responde a cualquier mensaje. respond_to? aparece usualmente en conexión con la condicion lógica if.

```
1 class Perro
2   def initialize(raza, nombre)
3     @raza = raza
4     @nombre = nombre
5   end
6 end
7
8 p = Perro.new('Alsacian', 'Lassie')
9 puts p.class.to_s
```

El resultado es:

```
RubyMate r8136 runnin Ruby r1.8.6
(/usr/local/bin/ruby)
>>> perro2.rb
```

```
perro
```

instance_of? regresa true si el objeto es instancia de la clase especificada.

1 num = 10 2 puts (num.instance_of? Fixnum) # true

Las clases abiertas de Ruby

En Ruby, las clases nunca son cerradas: siempre puedes agregar métodos a una clase que ya existe. Lo anterior aplica tanto para las clases que tu escribes como para las clases estándar incluídas en Ruby. Todo lo que tienes que hacer es abrir la definición de una clase existente y el nuevo contenido que especifiques va a ser agregado. En el ejemplo pasado, definimos la clase Motocicleta en el archivo p030motocicleta.rb.

```
1 class Motocicleta
2   def initialize(marca, color)
3     # Variables de instancia
4     @marca = marca
5     @color = color
6     end
7   def arranca
8     if (@engineState)
9       puts 'Motor encendido'
10     else
```

```
11       @engineState = true
12       puts 'Motor apagado'
13     end
14   end
15 end
```

Posteriormente, en el archivo p031pruebamotocicleta 'abrimos' la clase Motocicleta y definimos el método describe.

```
1 require 'p030motocicleta'
2 m = Motocicleta.new('Yamaha', 'rojo')
3 m.arranca
4
5 class Motocicleta
6   def describe
7     puts 'El color de la motocicleta es ' + @color
8     puts 'La marca de la motocicleta es ' + @marca
9   end
10 end
11 m.describe
12 m.arranca
13 puts self.class
14 puts self
15 puts Motocicleta.instance_methods(false).sort
```

Otro ejemplo: en el archivo p031babreperro.rb abrimos la clase Perro que definimos anteriormente en el archivo p029perro.rb y agregamos el método ladra_fuerte. (Observa que para poder abrir la clase, necesitamos requerirla primero):

```
1 # primero necesitamos requerir el archivo
2 # donde se encuentra la definicion de la clase
3 # perro
4 require 'p029perro.rb'
5
6 # abrimos de la clase y  definimos un método nuevo
7 class Perro
8   def ladra_fuerte
9     puts 'Woof! Woof!'
10   end
11 end
12
13 # inicializamos una instancia de la clase
14 d = Perro.new('Labrador', 'Benzy')
15
16 # podemos usar los metodos previamente definidos
17 # y el que acabamos de definir
18 d.ladra
19 d.ladra_fuerte
20 d.muestra
```

Si ejecutas este programa (y has seguido paso a paso los temas anteriores), el resultado tal vez no sea lo que tu esperas:

```
Intance_variables
is_a?
kind_of?
ladra
method
methods
muestra
nil
object_id
private_methods
protected_methods
public_methods
respond_to?
send
singleton_methods
taint
tainted?
to_a
to_plist
to_s
type
untaint

El id de objeto es 280360
Lo siento, el objeto no entiende el mensaje habla.
Ruff!Ruff!
Soy de la Raza Labrador y mi nombre es Benzy
Soy de la Raza Labrador y mi nombre es Benzy
Ruff!Ruff!
Woof!Woof!
Soy de la Raza Labrador y mi nombre es Benzy
```

¿Qué está pasando?, ¿Por qué esa larga lista de métodos si en el archivo que estamos ejecutando (p31abreperro.rb) sólo llamamos d.ladra, d.ladra_fuerte y d.muestra?.

La respuesta es simple pero importante: al inicio del programa estamos incluyendo el archivo p029perro, por lo tanto todo el codigo de ese archivo es ejecutado, no sólo la definición de la clase.

Observa que en el archivo p029perro.rb hacemos llamadas a métodos que imprimen a STDOUT (ed, llaman al método puts).

A continuación vemos otro ejemplo en donde agregamos un método a la clase String. (p032micadena).1

```
1 class String
2         def invierte
3                 puts self.reverse
4         end
5 end
```

```
6 cadena = "La ruta nos aporto otro paso natural!"
7 cadena.invierte
```

1 El método invierte no es en realidad muy útil, pero sirve para ilustrar el punto.

La facilidad que tenemos de abrir cualquier clase, incluso las clases built-in como String y Numeric es un tema medular en el estudio de Ruby y en la manera en que algunos programas en Ruby están escritos.

Rails hace uso extenso de esta característica de Ruby en el módulo ActiveSupport. De hecho, cuando eres nuevo en Ruby y en Rails, puede ser confuso identificar qué métodos son estándar en Ruby y qué métodos son definidos por Rails.

Por ejemplo, en el archivo lib/core_ext/integer/even_odd.rb, dentro de ActiveSupport, se encuentra la definición del el método multiple_of? dentro del módulo EvenOdd. (En realidad, el módulo EvenOdd está anidado dentro de otros módulos pero para simplificar el ejemplo, eliminamos la ruta completa):

```
1 module EvenOdd
2   def multiple_of?(number)
3     self % number == 0
4   end
5 end
6
```

Nota que hasta aquí, no se ha abierto ninguna clase de Ruby. Sin embargo, en el archivo lib/core_ext/integer/integer.rb encontramos:

```
1 # primero el archivo even_odd es requierido
2 require 'active_support/core_ext/integer/even_odd'
3
4 # se a bre la clase Integer
5 class Integer
6   # se incluye el módulo Integer
7   # recuerda que para  sipmlificar el ejemplo,
8   # eliminamos el identificador completo del módulo
9   # EvenOdd
10   include EvenOdd
11 end
12
13 # Toma en cuenta que esto es esencialmente lo  mismo que
14 # escribir:
15 #
16 #
17 # class Integer
18 #    def multiple_of?(number)
19 #     self % number == 0
20 #   end
21 # end
22 #
23 # Los metodos definidos en un modulo
24 # son agregados como metodos de instancia
25 # a cualquier clase en la que incluyamos dicho
26 # modulo usando el keyword include. (mixins)
```

```
27
28
```

ActiveSupport **es un ejemplo excelente de cómo podemos abrir las clases en Ruby y agregar métodos. Si quieres explorar más en este tema, analiza la librería Facets que contiene muchos ejemplos útiles, incluyendo algunos de los que se usan en Rails (en particular Facets/CORE).**

Constructores Literales

Esto significa que puedes usar notación especial en lugar de llamar new para crear objetos de una clase determinada. Las clases con constructores literales se muestran en nla siguiente tabla. Cuando usas uno de estos constructores literales, estas creando un nuevo objeto:

```
Clase      Constructor      Ejemplo
String      ""       "nueva cadena" o 'nueva cadena'
Symbol      :       :simbolo o :"simbolo con espacios"
Array      []       [1,2,3,4,5]
Hash      {}       {"Nueva York" => "NY", "Oregon" => "OR"}
Range      .. o ...      0...10 o 0..9
Regexp      //       /([a-z]+)/
```

Recolección de basura

La expresión p = nil significa que no hace referencia a nada. Si ahora escribimos p1 = nil, entonces el objeto Perro es abandonado y es elegible para la recolección de basura. Garbage Collection (GC). En Ruby, el objeto llamado heap destina un mínimo de 8MB. El recolector de basura de Ruby es del tipo llamado mark-and-sweep. Ruby usa un mecanismo conservador del tipo mark and sweep. No hay garantía de que un objeto sea recolectado antes de que el programa termine.

Métodos de clase

Un método de clase es enviado es enviado a la clase misma, no a una instancia de la clase. Algunas operaciones que pertenecen a la clase no pueden ser llevadas a cabo por instancias individuales de esa clase. new es un ejemplo excelente.

Llamamos Perro.new porque, hasta que hayamos creado una instancia de clase Perro no podemos enviarle mensajes! Además, el trabajo de crear un nuevo objeto lógicamente pertenece a la clase. No tiene sentido que una instancia de la clase Perro cree a su vez instancias de su misma clase. Sin embargo, si tiene sentido que el proceso de creación de instancias se centralizado como una actividad de la clase Perro. Es de vital importancia entender que Dig.new es un método que podemos accessar a través de la clase pero ne de las instancias de la clase.

Los objetos individuales "Perro" (instancias de la clase Perro) no tienen acceso a este método. Una clase tiene sus propios métodos, su propio estado y su propia identidad y no los comparte con sus instancias.

La siguiente tabla muestra un ejemplo (tomado del libru Ruby for Rails) de la notación que se utiliza:

Notación Se refiere a Boleto#precio El método de instancia precio de la clase Boleto Boleto.mas_caro El método de clase mas_caro de la clase Boleto Boleto::mas_caro Otra manera de referirse al método de clase mas_caro

Cuando escribimos acerca de Ruby, el símbolo de número(#) es utilizado algunas veces para indicar en método de instancia -por ejemplo, decimos File.chmod para denotar el método de clase chmod de la clase File y File#chmod para referirnos al método de instancia que tiene el mismo nombre. Esta notación no es parte dela sintaxis de Ruby, es sólo parte del folklore.

Herencia

En ruby, la herencia se maneja herencia, pero no soporta multiple herencia, por lo que cada clase solo puede tener una superclase de la que herede comportamintos.

la sintaxis es la siguiente:

```
class ClasePadre
   def a_method
     puts 'b'
   end
 end

 class ClaseCualquiera < ClasePadre
# <  significa  hereda
    def another_method
      puts 'a'
    end
 end
```

en esta sentencia se declara una ClasePadre,con un método, luego ClaseCualquiera, hija de ClasePadre con otro método, si hacemos lo siguiente:

```
 instance = SomeClass.new
 instance.another_method
 instance.a_method
```

Salida

a b

vemos que se aplica la herencia de métodos correctamente al llamar desde una instancia hija un método de la clase padre

Polimorfismo

Si queremos redefinir un método de la clase padre en la hija, solo sobrescribimos el método que deseamos redefinir en la clase hija, de esta manera, dependiendo de el tipo de la instancia desde la cual llamamos al método, se utilizará la correspondiente definición

Ademas, para llamar a la clase padre en cualquier momento desde un hija utilizamos la palabra reservada super, sin embargo, si quieres acceder a un clase superior a el padre en la jerarquía de herencia(como un abuelo, digamos), no se puede directamente, sin embargo hay una salida puedes darles nombre afiliados a los métodos que deseas usar luego, como un alias!, de hecho es exactamente esto, veamos

```
class X
   def foo
     "hello"
   end
 end

 class Y < X
   alias xFoo foo
     def foo
       xFoo + "y"
```

```
    end
 end

 puts X.new.foo
 puts Y.new.foo
```

```
Salida
```

```
hello
helloy
```

Esquema de administración y separación de la Memoria

En Construcción...

Implementación de Co-Rutinas

El manejo de threads(Corrutinas) en ruby es muy sencillo, la creación de threads se basa en pasar un bloque a la creación de un objeto de la clase thread.

```
t1 = Thread.new do
 10.times do
  puts "hello from thread 1"
  sleep(0.2)
 end
end
t2 = Thread.new do
 10.times do
  puts "hello from thread 2"
  sleep(0.2)
 end
end
t1.join
t2.join
```

Para pasar parámetros se hace de la siguiente forma:

```
t1 = Thread.new(1) do |id|
 10.times do
  puts "hello from thread #{id}"
  sleep(0.2)
 end
end
t2 = Thread.new(2) do |id|
 10.times do
  puts "hello from thread #{id}"
  sleep(0.2)
 end
end
t1.join
t2.join
```

Para controlar las secciones críticas tan solo hay que cambiar la variable de clase critical a true:

```
counter = 0;
t1 = Thread.new do
 100000.times do
  Thread.critical = true
  counter += 1
  Thread.critical = false
 end
end
t2 = Thread.new do
 100000.times do
  Thread.critical = true
  counter -= 1
  Thread.critical = false
 end
end
t1.join
t2.join
puts counter
```

Manejo de Excepciones

Una excepción es una clase de objeto especial, una instancia de la clase Exception o de una clase descendiente de esa clase que representa una condición especial; indica que algo ha salido mal. Cuando esto ocurre, se genera una excepción. Por defecto, los programas Ruby terminan cuando una excepción ocurre pero es posible escribir código que maneje estas excepciones.

Generar una excepción significa detener la ejecución normal del programa y transferir el control al código que maneja la excepción en donde puedes ocuparte del problema que ha sido encontrado o bien, terminar la ejecución del programa. Una de estas dos opciones (solucionar el problema de alguna manera o detener la ejecución de programa) ocurre dependiendo de si has proporcionado una cláusula rescue. Si no has proporcionado dicha cláusula, el programa termina, por el contrario, si la cláusula existe, el control de ejecución fluye hacia esta.

Ruby tiene algunas clases predefinidas -descendientes de la clase Exception- que te ayudan a manejar errores que ocurren en tu programa.

El siguiente método genera una excepción cada vez que es llamado. El segundo mensaje nunca va a ser mostrado.

```
1 def genera_excepcion
2         puts 'Antes de la excepcion.'
3         raise 'Ha ocurrido un error'
4         puts 'Despues de la excepcion'
5 end
6
7 genera_excepcion
```

El método raise está definido en el módulo Kernel. Por defecto, raise genera una excepción de la clase RuntimeError. Para generar una excepción de una clase en específico, puedes pasar el nombre de la clase como argumento al método raise.

```
1 def inverse(x)
2   raise ArgumentError, 'El argumento no es un numero' unless x.is_a? Numeric
3   1.0 / x
4 end
5 puts inverse(2)
6 puts inverse('hola')
```

Manejando una excepción Para manejar excepciones (handle exceptions), incluímos el código que pueda generar una excepción en un bloque begin-end y usamos una o más cláusulas rescue para indicarle a Ruby los tipos de excepción que queremos manejar. Es importante notar que el cuerpo de la definición de un método es un bloque begin-end explícito; begin es omitido y todo el cuerpo de la definición del método está sujeto al manejo de excepciones hasta que aparezca la palabra end.

```
1 def genera_y_rescata
2   begin
3           puts 'Estoy antes de raise.'
4                 raise 'Ha ocurrido un error.'
5                 puts 'Estoy despues de raise.'
6         rescue
7           puts 'He sido rescatado.'
8         end
9         puts 'Estoy despues de begin.'
10 end
11 genera_y_rescata
```

Observa que el código interrumpido por la excepción nunca es ejecutado. Una vez que la excepción es rescatada, la ejecución continúa inmediatamente después del bloque begin que la generó.

Puedes apilar cláusulas rescue en un bloque begin-end. Las excepciones que no sean manejadas por una cláusula rescue fluirán hacia la siguiente:

```
1 begin
2   # ...
3 rescue UnTipoDeExcepcion
4   # ...
5 rescue OtroTipoDeExcepcion
6   # ..
7 else
8   # Otras excepciones
9 end
```

Para cada cláusula rescue en el bloque begin, Ruby compara la excepción generada con cada uno de los parámetros en turno. La ocurrencia tiene éxito si la excepción nombrada en la cláusula rescue es del mismo tipo que la excepción generada. El código en una cláusula else es ejecutado si el código en la expresiôn begin es ejecutado sin excepciones. Si una excepción ocurre, entonces la cláusula else no es ejecutada. El uso de una cláusula eles no es particularmente común en Ruby. Si quieres interrogar a una excepción rescatada, puedes asignar el objeto de clase Exception a una variable en la cláusula rescue, como se muestra en el programa p046excpvar.rb

```
1 begin
2   raise 'Una excepcion.'
3 rescue Exception => e
```

```
4   puts e.message
5         puts e.backtrace.inspect
6 end
```

La clase Exception define dos métodos que regresan detalles acerca de la excepción. El método message regresa una cadena que puede proporcionar detalles legibles acerca de lo que ocurrió mal. El otro método importante es backtrace. Este método regresa un array de cadenas que representa la pila de ejecución hasta el punto en que la excepción fue generada. Si necesitas garantizar que algún proceso es ejecutado al final de un bloque de código sin importar si se generó una excepción o no, puedes usar la cláusula ensure. ensure va al final de la última cláusula rescue y contiene un bloque de código que siempre va a ser ejecutado.

Excepciones con Manejo de Archivos

Ejemplo:

```
1 # Abrir un archivo y leer su contenido
2 # Nota que ya que está presente un bloque, el archivo
3 # es cerrado automaticamente cuando se termina la ejecucion
4 # del bloque
5 begin
6   File.open('p014estructuras.rb', 'r') do |f1|
7     while linea = f1.gets
8       puts linea
9     end
10   end
11
12   # Crer un archivo y escribir en el
13   File.open('prueba.txt', 'w') do |f2|
14     f2.puts "Creado desde un programa Ruby!"
15   end
16 rescue Exception => msg
17   # mostar el mensaje de error generado por el sistema
18   puts msg
19 end
```

Gramática en EBNF del Lenguaje

En Construccion

Características Especiales del Lenguaje

Caracteristicas de Ruby

-Todo es un objeto

En ruby, se combinan las capacidades de los lenguajes funcionales con las de los lenguajes imperativos orientados a objetos, de esta manera cada elemento en ruby tiene un compartimiento similar, pero personalizado para las conveniencias de cada usuario. Tanto asi que incluso se pueden escribir metodos a números que escribimos normalmente, es decir podemos hacer: 5.times print { “Ruby es lo mejor” }

-Bloques

Funcionalidad traspasada de los lenguajes funcionales, uno puede escribir un bloque de código llamado cláusula a cualquier método, para describir como debe actuar.

-La apariencia visual de Ruby

A pesar de que Ruby utiliza la puntuación muy limitadamente y se prefieren las palabras clave en inglés, se utiliza algo de puntuación para decorar el código. Ruby no necesita declaraciones de variables. Se utilizan convenciones simples para nombrar y determinar el alcance de las mismas.

♦ var puede ser una variable local.

♦ @var es una variable de instancia.

♦ $var es una variable global

Estos detalles mejoran la legibilidad permitiendo que el desarrollador identifique fácilmente los roles de las variables. También se hace innecesario el uso del molesto self. como prefijo de todos los miembros de instancia.

-Ademas… Ruby ademas posee otras funcionalidades para aclarar la vida los programadores, entre la cuales estan:

♦ Manejo de Excepciones

♦ Mark and Sweep Garbage collector

♦ Mucho soporte para comunicación con C, con un agradable API para utilizar Ruby desde C

♦ Manejo de multitareas (Threads) independiente del sistema operativo, puedes incluso usar threads en DOS!

♦ Gran Portabilidad, corre en toda clase de sistemas Linux/Unix, Windows, OS/2, dos, etc…

Ejemplos del Lenguaje

Ejemplo 1

```
puts "Hello World"
print 'Enter your name: '
name= gets
puts "Hello #{name}"
5.times { puts "Hola"  }
```

Ejemplo 2'

```
class Mamifero
 def respira
   puts "inhala y exhala"
 end
end
class Gato<Mamifero
 def habla
   puts "Meow"
 end
end
misifus = Gato.new
misifus.respira
misifus.habla
```

```
class Ave
 def acicala
   puts "Estoy limpiando mis plumas."
 end
 def vuela
   puts "Estoy volando."
```

```
 end
end
class Pinguino<Ave
 def vuela
   puts "Lo siento, prefiero nadar."
 end
end
class Aguila<Ave
 end
 puts "Pinguino"
 p = Pinguino.new
 p.acicala
 p.vuela
 puts "Aguila"
 a = Aguila.new
 a.acicala
 a.vuela


 begin
  File.open('p014estructuras.rb', 'r') do |f1|
    while linea = f1.gets
      puts linea
    end
  end

  # Crer un archivo y escribir en el
  File.open('prueba.txt', 'w') do |f2|
    f2.puts "Creado desde un programa Ruby!"
  end
 rescue Exception => msg
 # mostar el mensaje de error generado por el sistema
 puts msg
end
```

```
p014estructuras.rb
var = 5
if var > 4
 puts "La variable es mayor que 4"
 puts "Puedo tener muchas declaraciones aqui"
 if var == 5
   puts "Es posible tener if y else anidados"
 else
   puts "Too cool"
 end
else
 puts "La variable no es mayor que 4"
 puts "Puedo tener muchas declaraciones aqui"
```

```
end
# Loops
var = 0
while var < 10
 puts var.to_s
 var += 1
end
```

Enlaces Externos

- Ruby [2] Sitio principal de Ruby en Internet.
- Ruby central [3] Recursos sobre Ruby en Inglés.
- Rubíes [4] Comunidad hispana de usuarios de Ruby.
- Ruby Tutorial [5] tutorial en castellano de Ruby.
- Programming Ruby [6], David Thomas y Andrew Hunt. Libro libre
- Ruby Argentina [7] Comunidad Argentina de Ruby
- RubyLit [8] Comunidad del Litoral Argentino de Ruby
- Full Ruby on Rails Tutorial [9]
- NetBEANS IDE [10]
- Ruby Tutorial [11]
- Programa Hola Mundo! [12]
- Palabras Reservadas [13]

Referencias

[1] http://poignantguide.net/ruby/expansion-pak-1.html
[2] http://www.ruby-lang.org
[3] http://www.rubycentral.com
[4] http://ruby.org.es
[5] http://rubytutorial.wikidot.com/
[6] http://www.rubycentral.com/book/
[7] http://rubyargentina.soveran.com/home
[8] http://www.rubylit.com.ar/wiki/show/HomePage
[9] http://www.meshplex.org/wiki/Ruby/Ruby_on_Rails_programming_tutorials
[10] http://www.netbeans.org/features/ruby/index.html
[11] http://rubytutorial.wikidot.com/primer-programa
[12] http://www.ruby-lang.org/es/documentation/quickstart/
[13] http://todoruby.blogspot.com/2009/03/palabras-reservadas-en-ruby.html

Fuentes y contribuyentes del artículo

Programación en Ruby *Fuente*: http://es.wikibooks.org/w/index.php?oldid=131802 *Contribuyentes*: Grupo Ruby, Oleinad, 3 ediciones anónimas

Licencia