Chapter 17

Ruby: una revisión rápida

Lo primero que uno aprende en un nuevo lenguaje es imprimir el mensaje "Hello World" en consola. Entonces, empecemos con el primer ejemplo:

```
puts "Hello World!"
```

Uno puede ejecutar esta instrucción directamente en consola:

```
ruby -e 'puts "Hello Ruby!\n"'
Hello Ruby!
```

Asimismo, uno puede guardarlo en un archivo llamado hello.rb y ejecutarlo de la siguiente forma:

```
ruby hello.rb
Hello Ruby!
```

Para la primera clase del curso utilizaremos la segunda opción.

17.1 Comentarios

Para realizar un comentario de una sola linea se debe utilizar el caracter "#"

```
# Comentario de una linea
puts "Hello World!"
```

El comentario tambien puede ir al final de una instrucción

```
puts "Welcome to Ruby!"     # comentario
```

Los comentarios de mas de una linea deben ir entre las cadenas "=begin" y "=end", por ejemplo:

```
=begin
Comentario de multiple lineas
Se puede poner codigo en ruby dentro del comentario
El código dentro de los comentarios no es interpretado
=end
```

17.2 Aspectos Básicos

Durante el curso utilizaremos tipos de datos basicos como por ejemplo: enteros (integer), reales (float), cadenas (string), arreglos (array), mapas (hashes) y simbolos (symbol). En ruby es un lenguaje orientado a objeto donde hasta los datos basicos son objetos, es decir, instancia de una clase. Por ejemplo, los enteros son objetos instanciados a partir de la clase Number:

```
123456
-543
```

Las cadenas son definidas a traves de comillas simples o comillas dobles. Por ejemplo 'Texto' and "Texto", ambos crea un string. Para imprimir una cadena en consola es posible utilizar la funcion puts.

```
puts "Texto"
puts 'Texto'
```

En el ejemplo anterior ruby imprime dos veces la cadena "Texto". La diferencia en que crear una cadena permite (a) utilizar el caracter de escape "" y (b) interpolar la cadena con otros valores. Por ejemplo:

```
puts "texto\n"          # imprime"texto" y crea una nueva linea
puts "suma :#{1+2}"     # imprime "suma: 3"
puts 'suma :#{1+2}'     # imprime "suma :#{1+2}"
```

Para interpolar, se debe introducir la expresión a evaluar entre llaves utilizando el caracter "#" por delante. Ruby, primero evaluara la expresión, posteriormente concatenara el resultado.

Ruby cuenta con operadores aritmeticos y logicos para armar diferentes expresiones:

```
var1 = 20
var2 = 60
var3 = var1 + var2          # evalua a 80
var1 < 25 and var2 > 45     # evalua a verdad
```

Los operadores soportados son muy similares a la mayoria de los lenguajes.

17.3 Variables

Constantes. Las constantes en Ruby normalmente se escriben en puras mayusculas. Por ejemplo:

```
MYCONSTANT = "hello"
```

En la practica, una "constante" en Ruby puede cambiar de valor (aunque va contra el concepto), pero el interprete de Ruby realizará el cambio de valor pero arrojara una alerta. Por ejemplo, si intentamos asignar un nuevo valor a la constante MYCONSTANT.

```
MYCONSTANT = "hello"
MYCONSTANT = "hello2"
```

Ruby lanzara la siguiente alerta.

```
(irb):34: warning: already initialized constant MYCONSTANT
```

Declarando Variables. Ruby es un lenguaje con sistemas de tipos dinamicos, es decir, que no se especifica el tipo de dato que una variable podra contener. Por lo que la asignación no varia entre diferente tipo de dato.

```
var1 = Person.new
var2 = 230
var3 = "hola"
var2 = Array.new
```

También es posible asigna dos valores de tipo diferente en a una misma variable. Note que esto en la practica no es recomendable ya que puede causar bugs.

17.4 Funciones

A continuación puede ver un ejemplo de como se puede crear una funcion en Ruby. Esta función recibe dos numeros y retorna la suma de los mismos.

```
def sum (n1, n2)
     n1 + n2
end

sum( 3 , 4)               # devuelve 7
sum("cat", "dog")         # devuelve "catdog"
```

En ruby todas las funciones retornan un valor, la regla general, es que retornan el valor resultante de evaluar la ultima expresión dentro de la función. En este ejemplo, la función *sum*, retorna el resultado de evaluar la suma de *n1* y *n2*. Si tu no estas comodo con esta, puedes utilizar el keyword *return* para hacer explicito el valor que retorna la función. Por ejemplo:

```ruby
def multiply(val1, val2 )
     result = val1 * val2
     return result
end

value = multiply( 10, 20 )
puts value
```

Como vimos, en ruby los números y las cadenas también son objetos, por lo que en este ejemplo el + en realidad no es un operador es un método. Es más el + es un método polimorfico, ya que dependiendo del objeto que recibe el mensaje funcionara de forma diferente. Por ejemplo, si son cadenas el + concatenara las cadenas, y si son numeros el + los sumara.

A continuación otro ejemplo de función esta vez utilizando interpolación de cadenas.

```ruby
def say_goodnight(name)
    "Good night, #{name}"
end
puts say_goodnight('Ma')  # imprime Good night, Ma
```

En ruby los nombres de funciones y métodos pueden estar compuestos de cualquier caracter. Ademas, para llamar a un método no es necesario mandar los argumentos en parentesis, dicho esto, las siguiente expresiones son equivalentes:

```ruby
puts say_goodnight('Ma')      # imprime Good night, Ma
puts say_goodnight 'Ma'       # imprime Good night, Ma
```

Funciones anonimas como argumento. En ruby es posible enviar una función anonmia como argumento al final de la llamada. Considere el siguiente ejemplo:

```ruby
def call_block                              #función llamada call_block
     yield("hello",2)                       # ejecutando la función anonima
end
call_block { | s, n | puts s*n, "\n" }     # llamando a call_block y enviando una fu

# imprime hellohello
```

El ejemplo imprime dos veces la palabra hello, asi es, el operador * también es polimorfico.

17.5 Clases y Objetos

A continación podrá encontrar un ejemplo donde se crea un clase *BanckAccount* y dos metodos: *initialize* y *test_method*.

```
class BankAccount
   def initialize()
   end

   def test_method
        puts "The class is working"
   end
end
```

El metodo *initialize* es llamado justo después que ruby crea un objeto. Para crear un objeto se debe llamar al método de clase *new* de la siguiente forma:

```
account = BankAccount.new()
```

En el ejemplo anterior se puede notar que en Ruby tampoco es obligatorio utilizar parentesis para los argumentos al momento de declarar un método. Es es particularmente util para los métodos que no tienen argumentos. Y para llamar un método se utilizar el .

```
account.test_method  # imprime The class is working
```

Para identificar una variable como variable de instancia se debe utilizar el operador @.

```
class BankAccount
   def initialize (number)
      @accountNumber = number
   end

   def deposit(amount)
        @accountNumber = @accountNumber + amount
   end
   def withdraw(amount)
        @accountNumber = @accountNumber - amount
   end
   def print
       puts "balance: #{@accountNumber}"
   end
end

account = BankAccount.new(1324)
account.deposit(200)
account.withdraw(100)
```

El ejemplo anterior muestra la clase BankAccount con un atributo **?**. *Note que en Ruby no es necesario hacer explicito al inicio de la clase que atributos tiene. En este ejemplo,* amount *es una variable local y* @accountNumber_ una variable de instancia.

Visibilidad. Los atributos en ruby por defecto son protegidos, es decir, solo se los puede acceder desde la misma clase o desde las clases hijas. Por otro lado, los métodos tienen visibilidad pública por defecto, es decir, que los métodos pueden ser llamados desde cualquier clase del sistema.

Accesores. Como los atributos son protegidos, para acceder a su valor desde otras clases se puede crear métodos accesores:

```
class BankAccount
   def accountNumber= (number)
      @accountNumber = number
   end
   def accountNumber(number)
      return @accountNumber
   end
   ...
end
account= BankAccount.new(1223)
account.accountNumber= 3
account.accountNumber  # => 3
```

En el ejemplo anterior creamos unos accesores, a diferencia de otros lenguajes no utilizamos las palabras *set* o *get*, en particular, para la asignación utilizamos el simbolo = en el nombre. Ademas es importante recordar que en ruby no es obligatorio poner los parentesis para llamar a un método.

El parser de Ruby también es muy flexible que considera las siguientes expresiones equivalentes.

```
account.accountNumber= 3
account.accountNumber = 3
```

17.6 Herencia

Para explicar como se implementa herencia en Ruby utilizaremos en el ejemplo de una clase llama *Song* definida a continuación:

```
class Song
  def initialize(name, artist, duration)
      @name = name
      @artist = artist
      @duration = duration
  end
```

```
end
song = Song.new("Bicylops", "Fleck", 260)
```

Primero, mencionar que la clase *Song* hereda de la clase *Object* definida en el core de Ruby, todas las clases en realidad heredan por defecto de esta clase. Por esta razón cualquier clase creada ya soporta varios métodos, estos métodos son los que hereda de su clase padre *Object*, por ejemplo:

```
song.to_s  # que devuelve #<Song:0xe6c>
```

El método *to_s* esta definidio en la clase padre e devuelve la posición de mémoria del objeto en formato cadena (string). Por lo anterior, si escribo un método con el mismo nombre en la clase *Song* en realidad estaria haciendo sobre-escritura.

```
class Song
  def initialize(name, artist, duration)
      @name = name
      @artist = artist
      @duration = duration
  end
  def to_s
       "Song: #{@name}--#{@artist} (#{@duration})"
  end
end
song = Song.new("Bicylops", "Fleck", 260)
song.to_s
# devuelve "Song: Bicylops--Fleck (260 )"
```

Para heredar explicitamente de otra clase se debe utilizar el operador "<" como se ve a continuación:

```
class KaraokeSong < Song
  def initialize(name, artist, duration, lyrics)
    super(name, artist, duration)
    @lyrics = lyrics
  end
end
song = KaraokeSong.new("My Way", "Sinatra", 225, "And now, the...")
song.to_s
#devuelve "Song: My Way--Sinatra (225)"
```

La clase *KaraokeSong* hereda de la clase *Song* y le agrega un atributo adicional *lyrics*. Note que desde el método *initialize* se llama al constructor de la clase padre para que inicialize los atributos restantes de la clase.

La clase *KaraokeSong* también puede sobre-escribir el método *to_s*, por ejemplo:

```
class KaraokeSong < Song
  # Format as a string by appending lyrics to parent's to_s value.
```

```
  def to_s
    super + " [#{@lyrics}]"
  end
end
```

La clase anterior sobre escribe *to_s*, este método devolvera el mismo *string* que la clase *Song* pero agregando el atributo lyrics al final. Recuerdo que en programación orientada objeto el operador *super* sirve para ejecutar el método de la clase padre. En este caso en particular, *super* llama al método *to_s* de la clase padre y luego concatena un string al final.

Para la primera parte del curso no se utiliza mucho sobre-escritura, pero si en la seguna parte del curso. Se recomienda repasar los conceptos de herencia llevados en programación avanzada.

Variables y metodos de clase. Anteriormentes, se vio como crear variables de instancia (atributos) y métodos de instancia. En ruby on rails, a veces es necesario llamar a métodos de clase, por lo que a continuación explicamos como se crean y se utilizan los métodos de clase. El siguiente ejemplo, crea una variable de clase *plays* que tiene doble arroba al inicio, ademas, existe un método de clase *printPlaysReport* que tiene el keywork "self" al inicio.

```
class Song
  @@total_plays = 0
  def initialize(name, artist, duration)
    @name = name
    @artist = artist
    @duration = duration
    @plays = 0
  end
  def play
    @plays += 1
    @@total_plays += 1
  end
  def printReport
    puts "this song play #{@plays} times"
  end
  def self.printPlaysReport
    puts "all songs play #{@@total_plays} times"
  end
end
```

El código anterior tiene dos variables *plays* y *total_plays* la primera contara cuantas veces un objeto en particular reprodujo una canción. Por otro lado, *total_plays* contara cuantas canciones se reprodujeron en total entre todos los objetos *Song*. Por ejemplo:

```
song1 = Song.new
song1.play
song2 = Song.new
song2.play

song1.printReport
# this song play 1 times
song2.printReport
# this song play 1 times
Song.printPlaysReport
# all songs play 2 times
```

Note que el el método *printPlaysReport* se ejecuta sobre la clase *Song*, no asi sobre una instancia como song1 o song2. Desde un punto de vista, se podria decir que las variables de clase son variables compartidas entre todas las instancias de la clase.

17.7 Arreglos y Hashes

Los arreglos en ruby son instancias de la clase Array y las hashes (o mapas) son instancia de la clase Hash. Como en varios lenguajes uno puede crear un arreglo llamando a su constructor a través del operador new, o utilizando corchetes.

```
days_of_week = Array.new
days_of_week.empty?  # devuelve true
```

En el ejemplo anterior se crea un arreglo vacio (tamaño 0), asi mismo, se llama al método empty? de la clase Array, el mismo que retorna verdad (true). Otra cosa a mencionar es que en ruby los nombres de los métodos pueden tener caracteres especiales como '?'.

```
days_of_week = Array.new(7)
#[nil, nil, nil, nil, nil, nil, nil]
```

El ejemplo anterior crea un arreglo de tamaño 7, por defecto, ruby llena los 7 slots con nil. Es posible enviar como segundo argumento el valor por defecto con el cual quieres que se inicializen los slots.

```
days_of_week = Array.new(7, "today")
#["today", "today", "today", "today", "today", "today", "today"]
```

Finalmente, es posible crear el arreglo de forma explicita:

```
days_of_week = Array[ "Mon", "Tues", "Wed", "Thu", "Fri", "Sat", "Sun" ]
#["Mon", "Tues", "Wed", "Thu", "Fri", "Sat", "Sun"]
```

Como un arreglo es una instancia de la clase Array tu puedes ejecutar cualquier método de esta clase sobre un arreglo. Por ejemplo:

```
days_of_week.at(0)      # devuelve "Mon"
days_of_week.size       # devuelve 7
days_of_week.empty?     # devuelve false
```

Ademas ruby te ofrece un poco de azucar sintactico para acceder a los elementos del arreglo de forma similar a la que se hace n otros lengaujes.

```
days_of_week = [ "Mon", "Tue", "Wed", "Thu", "Fri" ]
days_of_week[0]       # devuelve "Mon"
days_of_week[1]       # devuelve "Tue"
```

Otro dato curioso es que el + tambien sirve para concatenar arreglos:

```
days1 = ["Mon", "Tue", "Wed"]
days2 = ["Thu", "Fri", "Sat", "Sun"]
days = days1 + days2
# ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
```

Finalmente para modificar el elemento de un arreglo se puede utilizar simplemente el operador =

```
colors = ["red", "green", "blue"]
# ["red", "green", "blue"]
colors[1] = "yellow"      # devuelve "yellow"
colors
# ["red", "yellow", "blue"]
```

Hashes En los arreglos se puede acceder a los valores en base a los indices. Las estructuras Hash nos permite almacenar pares de datos key-value y hacer operaciones con estos datos.

```
inst = { "a" => 1, "b" => 2 }
inst["a"]  # devuelve  1
inst["c"]  # devuelve  2

# al especificar 0 en el constructor, el Hash inicializa los valores por defecto en
inst = Hash.new(0)
inst["a"] # devuelve  0
inst["a"] += 1
inst["a"] # devuelve  1
```

17.8 Estructuras de Control

If then else. A continuación se muestra un ejemplo de if-then-else en ruby.

```
if count > 10
     puts "Try again"
elsif tries == 3
```

```
     puts "You lose"
else
     puts "Enter a number"
end
```

While. A continuación se muestra un ejemplo de while en ruby.

```
while weight < 100 and num_pallets <= 30
    pallet = next_pallet()
    weight += pallet.weight
    num_pallets += 1
end
```

Iteradores. Los iteradores en realidad son métodos que reciben una función anonima al final, como se vio en un inicio. Por ejemplo:

```
animals = ["ant", "bee", "cat", "dog", "elk"]
animals.each {|animal| puts animal }
```

Each es un método que llama a la función anonima una vez por cada elemento dentro del arreglo. A continuación se muestran varios ejemplos de iteradores utiles en ruby.

```
3.times { print "X " }
1.upto(5) {|i| print i, " " }
99.downto(95) {|i| print i, " " }
50.step(80, 5) {|i| print i, " " }
```

17.9 Convenciones

A continuación algunas convenciones de nombre importantes:

- Variables Locales, en minuscula o con _ de inicio. Ejemplos: name, fish_and_chips, _26.
- Variables globales, empiezan con $. Ejemplo: $debug
- Variables de instancia, empiezan con @.
- Variables de clase,empiezan con @@.
- Constante, todo en mayuscula. Ejemplo: SINGLE, PI
- Nombres de clase, mayuscula inicial. Ejemplo: MyClass, FeedPerMile

En algunos casos, algunas companias cuando una variable tiene dos palabras utilizan guion bajo. Las clases con mas de dos palabras, cada palabra debe tener mayuscula inicial. Recuerde que en Ruby es valido (sobre todo en los métodos) utilizar caracteres como *?*,*!* o =.

Nombres de Archivos Normalmente los nombres de los archivos son en pura minuscula y si son mas de dos palabras con un guion bajo para separar las palabras, por ejemplo, hard_drive.rb

17.10 Material Adicional

- Documentación de Ruby: https://www.ruby-lang.org/es/documentation/
- Prueba Ruby en tu navegador: https://try.ruby-lang.org/
- Aprende a programar: https://pine.fm/LearnToProgram/
- Otros libros: https://github.com/EbookFoundation/free-programming-books/blob/main/books/free-programming-books-langs.md#ruby