Simplificando el flujo

¡Comencemos!

¿Qué aprenderás?

- Simplificar el código de un script en Ruby utilizando buenas prácticas y la regla de condición en positivo.
- Conocer sintaxis de if ternario en Ruby.
- Refactorizar flujo condicional en Ruby.

¡Vamos con todo!

Variantes de IF

If en una línea (inline)

En Ruby, también es posible utilizar versiones cortas de la instrucción if y unless de la siguiente forma: action if condition. A este tipo de expresiones se le conoce como if inline por estar en la misma línea.

Analicemos los siguientes ejemplos:

```
puts 'Ingresa tu edad: '
edad = gets.to_i

puts 'Eres mayor de edad!' if edad >= 18

puts 'Ingresa tu nombre: '
nombre = gets.chomp

puts "Hola! #{nombre}!" if nombre != ''
```

Esta forma de la instrucción if es más limitada ya que no tenemos manejo del flujo cuando no se cumple la condición (else if y else), sin embargo, es una muy buena solución cuando nos enfrentamos a evaluaciones sencillas como las anteriores.

Operador ternario

El operador ternario es una variante de if que permite operar en base a dos caminos en condiciones simples.

La lógica es la siguiente:

```
si_es_verdadero ? entonces_esto : sino_esto
```

```
edad = 18

puts edad >= 18 ? "Mayor de edad" : "Menor de edad"
```

Lo anterior se lee: Si la edad es mayor o igual a 18, imprime "Mayor de edad"; sino, imprime "Menor de edad".

Refactorizar

Cuando comenzamos a operar con condicionales es común que caigamos en redundancias innecesarias. Analicemos el siguiente ejemplo:

```
mayor_de_edad = true
zurdo = false

if mayor_de_edad == true
 if zurdo == true
   puts "Mayor de edad y zurdo!"
 else
   puts "Mayor de edad pero no zurdo!"
 end
else
 if zurdo == true
   puts "Menor de edad y zurdo!"
 else
   puts "Menor de edad pero no zurdo!"
 end
end
Reemplacemos los if anidados por condiciones múltiples:

if mayor_de_edad == true && zurdo == true
 puts "Mayor de edad y zurdo!"
elsif mayor_de_edad == true && zurdo == false
 puts "Mayor de edad pero no zurdo!"
elsif mayor_de_edad == false && zurdo == true
 puts "Menor de edad y zurdo!"
else
 puts "Menor de edad y no zurdo!"
end
```

Ahora podemos refactorizar las comparaciones en los condicionales que son innecesarias.

Recordemos que la instrucción if espera que el resultado se evalúe como true o false. Por lo tanto:

```
mayor_de_edad = true

if mayor_de_edad == true
 puts "Mayor de edad"
end
```

Es lo mismo que:

```
mayor_de_edad = true

if mayor_de_edad
 puts "Mayor de edad"
end
```

La comparación mayor_de_edad == true es redundante ya que la variable por sí sola se puede evaluar como true o false. Apliquemos esto en el ejemplo:

```
mayor_de_edad = true
zurdo = false

if mayor_de_edad && zurdo
 puts "Mayor de edad y zurdo!"
elsif mayor_de_edad && zurdo == false
 puts "Mayor de edad pero no zurdo!"
elsif mayor_de_edad == false && zurdo
 puts "Menor de edad y zurdo!"
else
 puts "Menor de edad y no zurdo!"
end
```

Podemos también refactorizar la comparación para evaluar que una variable booleana sea falsa, simplemente negando la condición:

```
mayor_de_edad = true
zurdo = false

if mayor_de_edad && zurdo
 puts "Mayor de edad y zurdo!"
elsif mayor_de_edad && !zurdo
 puts "Mayor de edad pero no zurdo!"
elsif !mayor_de_edad && zurdo
 puts "Menor de edad y zurdo!"
else
 puts "Menor de edad y no zurdo!"
end
```

Análisis léxico

En el colegio, en más de alguna ocasión, tuvimos que estudiar la estructura de una oración: Separar el sujeto del predicado, diferenciar los tiempos verbales y todo esto con tal de entender el significado de la oración.

La forma en que un computador lee nuestro código es exactamente igual. Existe un programa encargado de traducir cada una de nuestras instrucciones a un lenguaje que nuestro computador pueda entender.

Conocer los procesos que realiza Ruby al leer un programa nos permitirá entender el por qué son necesarias estas reglas y nos ayudará a memorizarlas.

¿Qué sucede cuando ingresamos al terminal y escribimos ruby mi_programa.rb?

```
Ruby lee y procesa nuestro código.
```

Detrás de esta lectura y procesamiento se están llevando a cabo una serie de procesos muy interesantes.

No necesitamos conocer estos procesos para poder construir programas en Ruby, sin embargo, conocerlos facilitará nuestra comprensión y nos ayudará a entender de mejor manera los mensajes de error que podemos obtener al ejecutar nuestro código.

El proceso

El proceso, desde la lectura hasta la ejecución, involucra una serie de etapas que estudiaremos a continuación:

- Tokenización.
- Lexing.
- Parsing.

Etapa 1: Tokenización

Ruby, al leer un código, lo primero que realiza es una tokenización. Esto consiste en separar cada palabra o símbolo en unidades llamadas tokens.

Por ejemplo al leer:

```
x = 10
```

Separa la instrucción en 3 tokens:

- `x`.
- `=`.
- `10`.

Etapa 2: Lexing

A través de un proceso llamado lexing se clasifican los tokens. Cada token es identificado según reglas.

Por ejemplo, como x no tiene comillas, Ruby sabe que corresponde a una variable.

Etapa 3: Parsing

La tercera y última etapa consiste en generar un árbol llamado árbol de sintaxis abstracta. Esto corresponde a una forma de representar el código que le permite -a Ruby- saber en qué orden ejecutar las operaciones, o descubrir si todo paréntesis abierto se cierra.

Al terminar la etapa de parsing se genera código bytecode. Esto corresponde al código que una máquina virtual Ruby puede leer. La máquina virtual utilizada por defecto se llama YARV (Yet Another Ruby Virtual Machine) y finalmente es esta la que ejecuta nuestro código.

Reglas Básicas

Ahora estudiaremos las reglas básicas de Ruby a partir de los tipos de tokens.

Mencionamos anteriormente que en una etapa los tokens son clasificados. Esta clasificación los separa en:

- Identificadores.
- Literales.
- Comentarios.
- Palabras reservadas.

Identificadores

Cuando hablamos de identificadores estamos hablando (para efecto de esta unidad) de nombres de variables y métodos.

Para que un nombre -de variable o método- sea válido tiene que seguir ciertas reglas. Un identificador válido es aquel que comienza con una letra de la a la z o un guión bajo (_). Puede ser seguido por letras, guión bajo o números, pero no puede empezar con un número. Los identificadores son sensibles a las mayúsculas. Esto quiere decir que "id" es distinto a "ID".

Por ejemplo, `1dia` es un ejemplo de identificador inválido porque comienza con un número. Evaluemos este identificador en IRB y veamos qué sucede:

```
1dia = '24 horas'
# SyntaxError: unexpected tIDENTIFIER, expecting end-of-input
```

El mensaje de error obtenido no puede ser más claro: tenemos un `tIDENTIFIER` (o sea un token identificador) no válido.

Otra regla importante es la sensibilidad a las mayúsculas. Esto quiere decir que un identificador que tenga una mayúscula en su nombre es distinto de uno que no la tiene:

```
gitHub_url = 'https://github.com/'
# => "https://github.com/"

puts github_url
# NameError (undefined local variable or method `github_url' for
main:Object)
# Did you mean?  gitHub_url
```

Nuevamente el mensaje de error es claro: No existe una variable o método github_url definido.

Literales

Los literales son los valores que asignamos a las variables o simplemente utilizamos para operar:

```
nombre = 'Chuck Berry'
```

'Chuck Berry' es un literal. Los literales se llaman así porque representan literalmente el valor.

Comentarios

Los comentarios, como ya hemos sabemos, corresponden a tokens que serán ignorados por YARV:

```
# Esta línea es un comentario
# Los comentarios son ignorados
```

```
puts 2 + 2 # También pueden existir contiguos a una línea de código
válida

# puts 2 + 3 (Esta línea también es ignorada)
```

Palabras reservadas

Corresponden a ciertas palabras que, como su nombre lo indica, están reservadas para la realización de ciertas tareas. Tienen un significado predefinido y no se pueden utilizar para nombrar variables.

Estas son:

BEGIN	class	ensure	nil	self	when
END	def	false	not	super	while
alias	defined	for	or	then	yield
and	do	if	redo	true	
begin	else	in	rescue	undef	
break	elsif	module	retry	unless	
case	end	next	return	until	

Imagen 1. Palabras reservadas
Fuente: Desafío Latam.

Y con esto ya sabemos como un computador puede leer el código de un lenguaje de programación.