Ruby Tutorial

...o como pasar un buen rato programando

Lección 1

- Introducción
- Instalación
- El Primer Programa
- Números en Ruby
- Strings y diversión
- Variables
- Alcance de las variables

Lección 2

- Introduciendo Datos
- Normas en los nombres

- Los métodos
- Los métodos: argumentos
- Rangos
- Arrays

Lección 3

- Bloques
- Más malabares con strings
- Expresiones Regulares
- Condicionales
- Bucles
- Números Aleatorios

Lección 4

- Clases y Objetos
- Accesores
- Ficheros: lectura/escritura
- Cargando librerías
- Herencia de clases
- Modificando clases
- Congelando objetos
- Serializando objetos

Lección 5

- Control de acceso
- Excepciones
- Módulos
- Constantes
- Hashes y Símbolos
- La clase Time

Lección 6

- self
- Duck Typing
- Azúcar Sintáctico
- Test de unidades

contacto

e-mail

Bloques

Un bloque es una porción de código encerrada entre paréntesis **{}** o entre **do…end**. Por lo tanto, un bloque es una forma de agrupar instrucciones, y solo puede aparecer después de usar un método: el bloque empieza en la misma línea que usa el método. El código dentro del bloque no es ejectuado en el instante que el intérprete de Ruby lo encuentra: Ruby se recordará del bloque (variables locales, …) y después entra en el método, ejecutando el bloque cuando es preciso.

Supongamos que existen dos métodos llamados greet1 y greet2:

```
#greet1, no necesita argumentos
greet1 {puts 'Hola'}

#greet2, necesita un argumento
greet2 ("argumento_cualquiera") {puts 'Hola'}
```

Lo usual es usar los paréntesis para bloques de una línea y el do…end para más de una línea.

yield

Un método puede usar el bloque mediante la palabra **`yield`**:

```
def metodo
  puts 'Comienzo del metodo'
  yield
  yield
  puts 'Final del metodo'
end

metodo{puts 'Dentro del bloque'}
```

La salida es:

```
'Comienzo del metodo'
'Dentro del bloque'     # primer yield
'Dentro del bloque'     # segundo yield
'Final del metodo'
```

Lo que sucede es que en el momento que el intérprete llega al yield, se ejecuta el código dentro del bloque, y luego se retorna al método.

Argumentos en los bloques

En los bloques se pueden usar argumentos especificándolos dentro de dos barras verticales | |. Y si se usan, en el yield no podemos olvidar darles valor:

```
def metodo
  yield('hola', 99)
end

metodo{|str,num| puts str + ' ' + num.to_s}  #hola 99
```

Un bloque de código devuelve un valor: el valor de la última expresión evaluada. Y este valor devuelto por yield, puede usarse dentro del método que invoca el bloque.

Los procs

Los bloques no son objetos, pero pueden convertirse en ellos gracia a la clase **`Proc`**. Los objetos tipo **proc** son bloques que se han unido a un conjuto de variables locales. Esto se hace gracias al método **`lambda`** del módulo Kernel.

```
prc = lambda{ "hola" }
```

Un bloque creado con lambda actúa como un método: si no especificas el número correcto de argumentos, no puedes llamar al bloque. La clase Proc tiene un método para llamar al bloque: el método **`call`**

```
prc = lambda {puts 'Hola'}
prc.call #llamamos al bloque

#otro ejemplo
toast = lambda do
  puts 'Gracias'
end
toast.call
```

La salida es:

```
Hola
Gracias
```

Para usar argumentos con lambda:

```
aBlock = lambda { |x| puts x }
aBlock.call 'Hola Mundo!'
```

La salida es:

```
Hola Mundo!
```

Los procs son muy útiles por que:

- No puedes pasar métodos dentro de otros métodos (usarlos como argumentos); pero si puedes usar procs como argumentos.
- Los métodos no pueden devolver otros métodos; pero sí pueden devolver un procs.

```
#uso de procs como argumentos

def metod1 proc1
  puts 'Principio del metodo'
  proc1.call
  puts 'Final del metodo'
end

hola = lambda do
  puts 'Hola'
end

metod1 hola
```

la salida es:

```
Principio del metodo
Hola
Final del metodo
```

Other interesting sites

Game Maker

Free Game Maker Tutorials & Game Maker Help Wiki

Fear no Evil - Rogue Trader

Though I walk through the valley of the shadow of death I fear no evil.

VisCom

Design, Create, Communicate!

exciting

exciting, all electron DFT