Tecnologías de Producción de Software 2025

Opción Ruby

Símbolos

Son como variables prefijados con : (dos puntos)

Ejemplos: **:action**, **:line_items**, **:+**

No es necesario declararlos

Se garantiza que son únicos

No es necesario asignarles ningún valor

```
:uno.object_id  # siempre devolverá lo mismo
"uno".object_id # siempre devolverá diferente
```

Colecciones

Arreglos

```
['Hola', 'Chau']

# sin interpolar
%w(Hola Chau #{2+2})

# interpolando
%W(Hola Chau #{2+2})

[1,2,3,4]
```

Hashes

```
# Versión 1.8
{
  :nombre    => 'Christian',
  :apellido    => 'Rodriguez'
}
# Versión > 1.8
{
  nombre:    'Christian',
  apellido:    'Rodriguez'
}
```

Expresiones Regulares

```
/^[a-zA-Z]+$/
"Do you like cats?" =~ /like/
"192.168.0.10" =~ /^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$/
```

Rangos

```
0..1
0..10
"a".."z"
"a"..."z"

# Pueden convertirse en arreglos
("a"..."z").to_a

# Rangos como intervalos
(1..10) === 5      # => true
(1..10) === 15     # => false
(1..10) === 3.1    # => true
```

Expresiones

Todo en Ruby devuelve un valor

```ruby
a = 3.14 # => 3.14

# Veamos el case
estado = nil
face =  case estado
        when "Feliz"  then ":)"
        when "Triste" then ":("
        else               ":|"
end
```

Bloques

Casi nunca vamos a utilizar un FOR o un WHILE

```ruby
3.times do |i|
  puts i
end

3.times { |i| puts i }
```

Bloques

Casi nunca vamos a utilizar un FOR o un WHILE

```ruby
3.times do |i|
  puts i
end

3.times { |i| puts i }
```

Bloques y colecciones

```ruby
# Selección de números pares
(1..10).select { |n| n.even? }

# Procesar cada elemento de una colección
(1..10).map { |n| n*2 }

# Calcular con los elementos de la colección:
(1..100).reduce { |sum,n| sum + n }
```

Bloques y colecciones

```ruby
# Selección de números pares
(1..10).select { |n| n.even? }
# o lo que es igual:
(1..10).select(& :even?)

# Procesar cada elemento de una colección
(1..10).map { |n| n*2 }
# o lo que es igual:
(1..10).collect { |n| n*2 }

# Calcular con los elementos de la colección:
(1..100).reduce { |sum,n| sum + n }
# o lo que es igual:
(1..100).reduce(:+)
```

Bloques y archivos

```ruby
File.open('/etc/passwd').each do |line|
  puts line if line =~ /root/
end
```

Freezando objetos

```
person1 = "Tim"
person2 = person1
person1.freeze
person2[0] = 'J'
```

OUTPUT

```
<anonymous>': eval:4:in `[]=': can't modify frozen String: "Tim" (FrozenError)
eval:4:in `<main>'
-e:in `eval'
```

Herencia

Permite crear clases que son un refinamiento o especialización de otra clase.
A esta clase se la llama *subclase* de la original.
A la clase original se la llama *superclase* de la subclase.
También se utilizan los términos: clase padre y clase hija.
El mecanismo de herencia es simple:

- Se heredan las capacidades del padre.
- Los métodos de instancia y clase del padre estarán disponibles en los hijos.

Ejemplo

```ruby
class Parent
  def say_hello
    puts "Hello from #{self}"
  end
end
```

```ruby
p = Parent.new
p.say_hello
```

```ruby
class Child < Parent
end
```

```ruby
c = Child.new
c.say_hello
```

Bloques e iteradores

Un bloque es código encerrado entre llaves o las palabras claves do y end.
Ambas formas son idénticas, salvo por la precedencia.
    Cuando el código del bloque entra en una línea usar {}.
    Cuando tiene más de una línea usar do / end.
Los bloques pueden verse como métodos anónimos.

Bloques e iteradores

Pueden recibir parámetros, que se explicitan entre barras verticales |.
El código de un bloque no se ejecuta cuando se define, sino que se almacenará para ser ejecutado más adelante.
En ruby, los bloques sólo podrán usarse después de la *invocación* de algún método:

- Si el método recibe parámetros, entonces aparecerá luego de ellos.
- Podría verse incluso como un parámetro extra que es pasado al método.

Ejemplo

```
sum = 0
[1, 2, 3, 4].each do |value|
  square = value * value
  sum += square
end
puts sum
```

- El bloque se invoca para cada elemento en el arreglo
- El elemento del arreglo es pasado al bloque en la variable `value`
- La variable `sum` declarada fuera del bloque es actualizada dentro del bloque

Bloques

- **Regla importante:** *si existe una variable en el bloque con el mismo nombre que una variable dentro del alcance pero creada fuera del bloque, ambas serán la misma variable. En el ejemplo hay sólo una variable* `sum`
- Veremos que el comportamiento mencionado podremos cambiarlo
- Si una variable aparece sólo en el bloque, entonces será local al mismo (como `square`)

Bloques

```ruby
# assume Shape defined elsewhere
square = Shape.new(sides: 4)
#
# .. lots of code
#
sum = 0
[1, 2, 3, 4].each do |value|
  square = value * value
  sum += square
end

puts sum
square.draw # BOOM!
```

Bloques

Podemos solucionar el problema de square

```ruby
square = "some shape"
sum = 0
[1, 2, 3, 4].each do |value; square|
  square = value * value # different variable
  sum += square
end
puts sum
puts square
```

Bloques

- Mencionamos que los bloques se utilizan de forma adyacente a la llamada a un método y que no se ejecutan en el momento en que aparecen en el código.
- Para lograr este comportamiento, dentro de un método cualquiera, podremos invocar un bloque:
  - Los bloques se invocarán como si fueran métodos.
  - Para invocar un bloque se utiliza la sentencia `yield`.
  - Al invocar `yield` ruby invocará al código del bloque .
  - Cuando el bloque finaliza, ruby devuelve el código inmediatamente al finalizar el llamado a `yield`.

Bloques

```ruby
def three_times
  yield
  yield
  yield
end

three_times { puts "Hola" }
```