Sun*

Methods

Outline

1. Method introduction
2. Method arguments
3. Method returning value
4. Class method and instance method
5. Block, Proc, Lambda

1. Method introduction

- ❖ Ruby methods are very similar to functions in any other programming languages.
- ❖ Method has a name, take some input, do something with it, and return a result.
- ❖ Method name should begin with a lowercase letter.
- ❖ Execute method by calling method via object.

Sun*

1. Method introduction

Syntax:

```
def method_name [([arg [= default]]...[, * arg [, &expr ]])]
  expr..
end

def method_name([arg [= default]]...[, * arg [, &expr ]]) = expr..
```

```
#Example
def print_your_name(name)
  puts "Your name is " + name
  puts "Another name"
end
def square(x) = x * x
```

Sun*

2. Method arguments

```ruby
#Example
def calculate_value_1(x,y)
  p x + y
end

def calculate_value_2(value='default', arr=[])
  puts value
  puts arr.sum
end

def calculate_value_3(x,y,*otherValues)
  puts otherValues
end
```

```ruby
#Excute method
calculate_value_1(1, 2)
calculate_value_2
calculate_value_2(1)
calculate_value_3(1, 2)
calculate_value_3(1, 2, 4, true)
calculate_value_4(1, 2)
calculate_value_4(1, 2, c: 3)
calculate_value_4(1, 2, c: 3, d: 4)
```

Sun*

3. Method returning value

- ❖ Methods return the value of the last statement executed.
- ❖ An explicit return statement can also be used to return from function with a value, prior to the end of the function declaration.

3. Method returning value

```ruby
#Example
def calculate_value(x,y)
   p "x / y = #{x / y}"
end

def second_calculate_value(x,y)
  return puts " x / y = #{x / y}"
  puts " End line second_calculate_value method"
end

def third_calculate_value(x,y)
  return puts " x / y = #{x / y}" if y > 0
  puts " Don't calculate because y <= 0"
end

def fourth_calculate_value(x,y)
  return puts " x / y = #{x / y}" if y > 0
end
```

```ruby
#Excute method
puts "1.Call method calculate_value(x,y)"
calculate_value(4, 2)

puts "2.Call method second_calculate_value(x,y)"
second_calculate_value(4, 2)

puts "3.Call method third_calculate_value(x,y)"
third_calculate_value(1, 0)

puts "4.Call method fourth_calculate_value(x,y)"
fourth_calculate_value(1, 0)
puts " fourth_calculate_value(1, 0) would be return
nil"
```

Sun*

4. Class method and instance method

```ruby
#Example
class Invoice
  # class method
  def self.print_out
    "Printed out invoice"
  end

  # instance method
  def convert_to_pdf
    "Converted to PD"
  end
end

puts "1.Execute class method"
puts Invoice.print_out
puts "2.Execute instance method"
puts Invoice.new.convert_to_pdf
```

4. Class method and instance method

```
#Result
1.Execute class method
Printed out invoice
2.Execute instance method
Converted to PD
```

Sun*

5. Block, Proc, Lambda

5.1. Block

- Blocks are enclosed in a do /end statement or between brackets {}, and they can have multiple arguments.
- The argument names are defined between two pipe | characters.
- The use of blocks is fundamental to the use of iterators.

```
#Example
1.upto(10){|x| puts x}
1.upto(10) do |x|
  puts x
end
1.upto(10)     # No block specified
{|x| puts x}  # Syntax error: block not after an invocation
```

Sun*

5. Block, Proc, Lambda

5.1. Block

❖ Implicit block: Ruby methods can implicitly take a block, without needing to specify this in the parameter list.

```ruby
#Example
def hello(&block)
  yield
end

hello do
  puts " Implicit block"
end
```

```
#Result
Implicit block
```

5. Block, Proc, Lambda

5.1. Block

```ruby
#Example
def hello(&block)
  yield
end

hello do
  puts " Implicit block"
end
```

Explicit block:

- ❖ Ruby allows to pass any object to a method and have the method attempt to use this object as its block. If we put an ampersand in front of the last parameter to a method, Ruby will try to treat this parameter as the block method.
- ❖ When we write our method definition, we can explicitly state that we expect this method to possibly take a block. Ruby uses the ampersand for this as well.
- ❖ If the parameter is already a Proc object, Ruby will simply associate it with the method as its block. If the parameter is not a Proc, Ruby will try to convert it into one (*by calling to_proc on it*) before associating it with the method as its block.
- ❖ block is a Proc object, instead of yielding to it, we can call it.

Sun*

5. Block, Proc, Lambda

5.2. Proc

A "proc" is an instance of the Proc class, which holds a code block to be executed, and can be stored in a variable. To create a proc, you call Proc.new and pass it a block.

```
#Example
# A block is just a Proc!
def what_am_i(&block)
  block.class
end
puts what_am_i {}
# => Proc
---------------------------------------

square = Proc.new do |n|
  n ** 2
end
square.call (2)

#Result
4
```

5. Block, Proc, Lambda

5.3. Lambda

Lambda is an anonymous function:

- ❖ It has no name (hence anonymous)
- ❖ Used when you don't want the overhead/formality of a normal function
- ❖ Is not explicitly referenced more than once, unless passed as an argument to another function

```ruby
#Example
puts "1. Execute square"
square = lambda {|n| n ** 2}
puts " 2**2 = #{square.call (2)}"
```

```
#Result
1. Execute square
 2**2 = 4
```

5. Block, Proc, Lambda

5.4. Proc vs Lambda

- Both of them are instance of Proc class
- Lambdas check the number of arguments, while procs do not

```
#Example
lam = lambda { |x| puts x }     # creates a lambda that takes 1 argument
lam.call(2)                     # prints out 2
lam.call                        # ArgumentError: wrong number of arguments (0 for 1)
lam.call(1,2,3)                 # ArgumentError: wrong number of arguments (3 for 1)

proc = Proc.new { |x| puts x } # creates a proc that takes 1 argument
proc.call(2)                    # prints out 2
proc.call                       # returns nil
proc.call(1,2,3)                # prints out 1 and forgets about the extra arguments
```

Sun*

5. Block, Proc, Lambda

5.4. Proc vs Lambda

- ❖ Lambdas and procs treat the ‘return’ keyword differently:
  - ➔ ‘return’ inside of a lambda triggers the code right outside of the lambda code.
  - ➔ ‘return’ inside of a proc triggers the code outside of the method where the proc is being executed

Sun*

5. Block, Proc, Lambda

5.4. Proc vs Lambda

```ruby
#Example
def lambda_test
  lam = lambda{return}
  lam.call
  puts "End line of lambda_test method"
end

def proc_test
  proc = Proc.new{return puts "Return in proc"}
  proc.call
  puts "End line of proc_test method"
end

puts "1. Execute lambda_test"
lambda_test
puts "2. Execute proc_test"
proc_test
```

```
#Result
1. Execute lambda_test
End line of lambda_test method
2. Execute proc_test
Return in proc
```

5. Block, Proc, Lambda

5.5. Summary differences

- ❖ Procs are objects, blocks are not.
- ❖ At most one block can appear in an argument list.
- ❖ Lambdas check the number of arguments, while procs do not.
- ❖ Lambdas and procs treat the ‘*return*’ keyword differently.

Sun*

References

- http://ruby-doc.org/core-3.1.0/doc/syntax/methods_rdoc.html
- https://www.tutorialspoint.com/ruby/ruby_methods.htm
- https://github.com/awesome-academy/RubyExample_TFW

Sun*

Question & Answer?

Sun*