Method:

Ruby methods are very similar to functions in other programming language.

Ruby methods are used to bundle one or more repeatable statements into a single unit.

#Simple method

def hello

puts 'Hello'

end

hello =># Hello

Note: Def, end are keywords and hello is method name(small letters)

and (In Real time) in ruby: function is a method but in rails method is action in controller.

# Pass one argument in method

def hello(name)

puts 'Dear ' + name

end

hello('satish')

#pass number of arguments

def my(\*value)

puts "My name is #{value[0]}"

puts "My age is #{value[1]}"

puts "My sal is #{value[2]}"

puts "My job is #{value[3]}"

end

puts my(32, "srinivas", 100.00, :IT)

#def some\_method(a, b=5, \*p) - correct

#def some\_method(a, \*p, b=5) - incorrect!!!

# pass arguments values direct

def my(arg1="Divya", arg2="Shashank", arg3="Srinivas")

puts "#{arg1}, #{arg2}, #{arg3}."

end

puts my

puts my("ruby", "Rails")

def check\_sign(number)

if number > 0

"#{number} is positive"

else

"#{number} is negative"

end

end

Iterators:Iterators are nothing but methods supported by *collections*.

In Ruby, arrays and hashes, ranges can be termed collections.

Iterators return all the elements of a collection, one after the other.

Iterator methods :

each, map/collect, select, times, upto, downto, step

Note: We have to use iterator on behalf of loops(for or while) statements for best performance.

Practice Enumerable

**Syntax:**

collections.each do |abc|  
 puts abc  
end

**OR**

collections.each { |abc| puts abc }

Examples:1

sites = ["net", "co.in", "org"]

sites.map do |site|

puts "www.yahoo." << site

end

Examples:2

|  |  |
| --- | --- |
| sites = ["in", "co.in", "edu"]  sites.each { |site|  puts "www.mywebsite.#{site}"  } | my\_fruits = ["apple", "banana", "cherry"]  my\_fruits.each do |fruit|  puts "I love #{fruit}".upcase  end |

Examples:3

|  |  |
| --- | --- |
| 5.times do  print "Hello! "  end | 15.times do |i|  puts "Loop number #{i}"  end |
| 0.upto(9) do |x|  puts x end | 1.upto(10) { |i| print i, " " } |
| 0.step(12, 3) {|x| print x, " " } | 1.step(10, 2) { |i| print "#{i} "} |
| 10.downto(0) { |n| print n, ".. " } | 20.downto(0) { |n| print n, ".. " } |
| array = [10,20,30,40,50,60,70,80,90,100]  array.each\_index {|i| puts "#{array[i]} "} | a = 1..20 a.each{|x| puts "\*" \* x} |
| data =[1,2,3,4]  data.collect {|x| puts x\*\*x} | words = %w[earth jupiter venus mercury Saturn]  words.map {|word| puts word.upcase} |

Block:Block is same such as method, but Block consists chunks of code(group of code).

We must use { } with yield keyword.

#Examples:1

def call\_block

puts 'Start of method'

yield

puts 'End of method'

end

puts call\_block {puts 'In the block'}

#Example:2

def i\_am\_block

yield

end

puts i\_am\_block {puts 'Static or Dynamic content display here..This is nothing but content on page in rails….'}

Note:We have to pass any argument in Yield statement from outside of block.

In Rails: Yield is equal to View(page content)

Proc & Lambda

procs and lambdas define blocks of code that can be executed when called.

Procs and lambdas are generally used for the same purpose, but they have some differences between them:

Blocks, Procs and lambdas are all closures.

Proc:Proc objects are blocks of code. We can store a Proc in a variable and call with call method.

Note: Proc supports new keyword, no of argument values and return value.

#Example1

lamb = lambda {|x, y| puts x + y}  
pro = Proc.new {|x, y| puts x + y}  
  
# works fine, printing 6  
pro.call(2, 4, 11)  
  
# throws an ArgumentError  
lamb.call(2, 4, 11)

So, when should you use blocks over Procs? My logic is as follows:

1. Block: in application.html.erb(Yield) or other layouts files.
2. Proc: You want to reuse a block of code multiple times.

Lambda: Lambdas are almost identical to Procs but with two key differences.

We can store a Lambda in a variable and call with call method

Note: Lambda doesn’t support new keyword, no of arguments and return value.

We must pass equal argument in lambda.

|  |  |
| --- | --- |
| Method:  def test\_blocks  puts "in test\_blocks"  end  puts test\_blocks | Block:  def test\_blocks  puts "in test\_blocks"  yield  end    test\_blocks { puts "in the block" } |
| my\_proc = Proc.new{puts "This is my Proc Object"}    my\_proc.call    Proc:  def test\_blocks(po)  puts po.call  end    p = Proc.new { puts "in the Proc !" }  test\_blocks(p)  #in the Proc!  Pass and Checking Arguments in proc  p = Proc.new {|a, b| puts a\*\*2 + b\*\*2 }  p.call(1, 2) # => 5  p.call 1 # => NoMethodError: undefined method `\*\*' for nil:NilClass  p.call 1, 2, 3 # => 5 | my\_lambda = lambda{puts "This is my Lambda Object"}    my\_lambda.call  Lambda:  def test\_blocks(la)  puts la.call  end    l = lambda { puts "in the Lambda !" }  test\_blocks(l)  # Output: # in the Lambda !  Pass and Checking Arguments in lambda  l = lambda {|a, b| puts a\*\*2 + b\*\*2 }  l.call(1, 2 )# => 5  l.call 1 # => ArgumentError: wrong number of arguments (1 for 2)  l.call 1, 2, 3 # => ArgumentError: wrong number of arguments (3 for 2) |
| Check Proc return value  def bar1  f = Proc.new { return "I am proc and accept return value" }  f.call  return "I am not accept return value"  end  puts bar1=>  I am proc and accept return value | Check Lambda return value  def bar  f = lambda { return "I am lambda and not accept return value" }  f.call  return "I am Ordinary return value"  return "I am Ordinary return value1"  end  puts bar=> I am Ordinary return value" |
| When to Use in rails   * If we want to save a piece of code in a variable for reuse then use an explicit **Proc** object. * Use in model |  |