# Методы, блоки, yield, return, lambda, proc

### ****Методы****

Объявление метода происходит при помощи выражения **def** после которого следует имя метода и перечень аргументов необходимых для него, пример:

def sum(a, b)

  puts "#{a} + #{b} = #{a+b}"

end

sum(1, 9)

Для методов не нужно явно указывать возвращаемое значение, они вернут результат выполнения посленей строки своего кода, однако иногда полезно явно указывать возвращаемое значение при помощи выражения **return**. Это «иногда» наступает тогда, когда код слишком сложен и сложно сразу определить, что будет возвращаться (например, когда метод содержит много условий). Еще **return** следует использовать тогда, когда вас не устаивает то, что метод возвращает результат выполнения последней строки. Пример использования **return**:

def sum(a, b)

  puts "#{a} + #{b} = #{a+b}"

  return a + b

end

sum(10, 5)

**Аргументы методов**  
Аргументы методов — это параметры, которые передаются в метод и с которыми метод выполняет определенные действия или из-за которых метод изменяет свое поведение. Аргументы бывают предопределенными (необязательными) и не предопределенными. Предопределенные аргументы используются тогда, когда, например значение аргумента в большинстве случаев одинаково, однако, пользователю метода может понадобится его изменить, непредопределенные аргументы были приведены в примерах выше. Когда вы используете непредопределенные аргументы метода, например в количестве 2, но при вызове метода передаете только один аргумент, то метод вернет ошибку: ArgumentError: wrong number of arguments (1 for 2), если же один из двух аргументов будет предопределенным, то значение автоматически присвоится тому аргументу, который является не предопределенным. Пример предопределенных аргументов:

def sum(a,b=5,c=10)

  puts "a = #{a}\nb = #{b}\nc = #{c}"

  puts a + b + c

end

sum(10)

#a = 10

#b = 5

#c = 10

#25

sum(10,15)

#a = 10

#b = 15

#c = 10

#35

sum(10,15,20)

#a = 10

#b = 15

#c = 20

#45

Обратите внимание на последовательность аргументов метода! Я сначала указываю не предопределенные аргументы, а затем предопределенные.Это делается не просто так, просто так легче понять, какой значение какому аргументу будет передано. Если вы будете использовать другой порядок следования аргументов, то присвоение аргументам значения будет происходить несколько неожиданно:

def sum(a=5,b=6,c)

  puts a,b,c

end

sum 2

#5

#6

#2

Методы в Ruby могут принимать не только предопределенное количество аргументов, но и произвольное их количество. Если быть совсем точным, то один из аргументов метода мы объявляем коллекцией значений при помощи оператора splat «\*». В методе может быть всего один такой аргумент. Пример:

def do\_something\_with\_collection(m,\*strings)

  result = []

  strings.each{|str| result << str.send(m)}

  return result

end

collection = %w{hello ruby readers}

#=> ["hello", "ruby", "readers"]

result\_collection = do\_something\_with\_collection(:upcase,\*collection)

#=> ["HELLO", "RUBY", "READERS"]

result\_collection = do\_something\_with\_collection(:downcase,\*result\_collection)

#=> ["hello", "ruby", "readers"]

При передаче аргументов в метод в виде массива мы также должны использовать оператор **\***, который превращает массив в набор значений, иначе аргументу strings будет определено одно значение — массив, что нас не устраивает. При создании методов с произвольным количеством аргументов следует обратить внемание на такое правило: аргумент реализующий прием множественного количества аргументов должен быть представлен последним в списке аргументов метода. Примеры:

def multisum(a,b=5,\*c)

  sum = c.inject{|sum, num| sum += num}

  sum\*a\*b

end

multisum(5, \*[1, 2, 3, 4, 5]) #=> 70

В метод на самом деле передается набор аргументов: 5,1,2,3,4,5, поэтому аргумент a получает значение 5, аргумент b получает значение 1, а аргумент с получает все остальные значения. При таком положении дел предопределенные аргументы становятся бесполезными, так как если метод получает произвольное количество параметров, то они могут быть переписаны. Выход из ситуации простой: не использовать предопределенные аргументы и аргументы получающие коллекцию значений в одном методе. Произвольное количество значений вы можете изначально передавать в виде коллекции. Пример:

def multisum(arg)

  sum = arg[:collection].inject{|sum, num| sum += num}

  sum \* arg[:a] \* arg[:b]

end

multisum({a:5, b:1, collection:[2,3,4,5]}) #=> 70

**Приемники методов**  
Метод пренадлежа объекту имеет приемник — объект, которому он принадлежит. Если метод вызывается без указания приемника, то это означает, что приемником является текущий объект, то есть тот, в контексте которого вызывается метод. Примеры:

class A

  def self.hello

    puts 'hello'

  end

  def bye

    puts 'bye'

  end

end

A.hello #hello

A.new.bye #bye Приемник **self** указывает на текущий объект, то есть метод .hello является методом класса, а метод #bye — методом объекта.

**Трюки с методами**

В Ruby существует возможность вызывать методы не на прямую, а через метод **#send** передавая ему в качестве аргумента символ соответствующий имени метода, пример:

Вызов:

hello".upcase #=> "HELLO"

…соответствует:

"hello".send(:upcase) #=> "HELLO"

Одной из сильных сторон Ruby является метапрограммирование, которое позволяет создавать методы непосредственно во время выполнения программы. Существует несколько вариантов создания методов «на лету» при помощи **.define\_method** в контексте класса, при помощи **#define\_singleton\_method** в контексте объекта и обработкой ошибок вызова несуществующего метода при помощи **#method\_missing**. Примеры:

class A

  define\_method :hello do

    puts 'hello'

  end

end

A.new.hello # hello

a = A.new

a.define\_singleton\_method(:bye) do |name|

  puts "Hello, #{name}"

end #=> #<Proc:0x86df814@(irb):18 (lambda)>

a.bye('Vladimir') #Hello, Vladimir

class A

  def method\_missing(name, arg, &block)

    if name =~ /count\_to/

      (arg + 1).times {|n| puts n}

    end

  end

end

a = A.new #=> #<A:0x86a0ba0>

a.count\_to(5)

#0

#1

#2

#3

#4

#5

.define\_method и #define\_singleton\_method принимают имя метода в виде строки или символа и блок кода, который превращается в процедуру, которая вызывается в контексте объявляемого метода. .define\_method и #define\_singleton\_method следует использовать только в случае реальной необходимости, когда, например, имя метода задается переменной. Во всех остальных случаях следует использовать стандартный синтаксис объявления методов при помощи def.

#method\_missing — используется для обработки ошибки **NoMethodError**, которая возвращается при попытке вызват несуществующий метод. То есть #method\_missing не объявляет методы, а обрабатывает ошибки. В контексте #method\_missing вы можете на основании имени вызываемого метода обработать ошибку, то есть симитировать существование метода или нескольких методов.

**singleton методы** — это индивидуальные методы объекта, которые хранятся в singleton классе объекта по той простой причине, что объекты не хранят методы. Подробней singleton классы рассматривались в предыдущей главе учебника.

**Блоки**

Блок — это произвольный код, который можно передать *любому* методу в качестве неявного последнего аргумента. Следует понимать, что при этом блок является особой конструкцией языка и обособлен от списка явных аргументов метода, что означает следующее:

* методу можно передать только один блок;
* он всегда идет в самом конце и вынесен за скобки с аргументами (если они есть);
* его можно задать при вызове любого метода, независимо от того, указаны в определении метода аргументы или нет.

Код может находиться внутри фигурных скобок { } или ключевых слов do end. Что именно использовать — решать вам, потому что обе конструкции описывают совершенно одинаковые по свойствам блоки. Обычно в фигурные скобки заключают код, состоящий из одной строки, при этом метод и его блок записывают на одной строке. Если код блока слишком длинный или объемный, вместо скобок на строке с вызовом метода оставляют do, а сам код пишут с новой строки с отступом и в конце «закрывают» его end'ом.

def say\_hello

puts "Поклон тебе, Цезарь!"

puts "Идущие на смерть приветствуют тебя!"

end

say\_hello { puts "Отпустите меня домой" }

Как видим, Ruby совершенно не возражает, что мы добавили к методу блок, хотя сам метод ничего о нем не знает!

**Передача контроля блоку**

Существует два способа выполнить код блока, переданный методу. В этой статье рассматривается только один из них — ключевое слово yield.

yield — это часть синтаксиса языка, как и объявление блока (поэтому ее нельзя переопределить). Когда в теле метода мы вызываем yield, то подразумеваем следующее: «отдай управление блоку, а когда тот закончит, верни управление этому методу».

Попробуйте переписать приветствующий метод так:

def say\_hello

puts "Поклон тебе, Цезарь!"

yield

puts "Идущие на смерть приветствуют тебя!"

end

и еще одна попытка передать блок:

say\_hello { puts "То, что мы делаем в жизни, отзывается в вечности." }

Поклон тебе, Цезарь!

То, что мы делаем в жизни, отзывается в вечности.

Идущие на смерть приветствуют тебя!

=> nil

Вуаля! Теперь попробуйте вызвать метод без блока.

say\_hello

LocalJumpError: no block given (yield)

Ошибка! Ruby теперь в обязательном порядке требует от нас блок! В некоторых случаях без блока действительно никак, но чаще всего логика метода вполне позволяет обойтись без него, как в этом случае. Поэтому перед вызовом беспощадного yield следует удостовериться, а есть ли блок, с помощью метода block\_given?:

def say\_hello

puts "Поклон тебе, Цезарь!"

yield if block\_given?

puts "Идущие на смерть приветствуют тебя!"

end

say\_hello

Поклон тебе, Цезарь!

Идущие на смерть приветствуют тебя!

=> nil

**Блок — это просто кусок кода**, который, обычно, присоединен к методу, процедуре или Lambda-функции. Важно запомнить, что блок кода — это не объект, это любой код, определенный внутри фигурных скобок { и }, или внутри контейнера do-end.

Код внутри блока это всего-лишь логика, которая еще не обернута в объект. Когда этот блок передается методу и в этом методе используется выражение **yield**, для использования блока, тогда Ruby обернет блок в специальный тип объекта. Этим типом объекта является **объект Proc**. Как только блок становится объектом Proc, Ruby может использовать его и код содержащийся в блоке оживает.

Таким образом объект Proc необходим для работы с блоками кода. Следующий вопрос: Зачем нам необходима lambda?

**Lambda — это разновидность объекта Proc**, лямбда почти полностью иддентична Proc если не обращать внимание на две ключевые особенности. Нижеприведенный пример показывает Proc и лямбду полученные из одного и того же суперкласса — Proc.

def what\_are\_lambdas\_and\_procs

  puts "--- #{\_\_method\_\_} ---"

  lam = lambda { puts 'lam variable assigned a lambda' }

  lam.call

  puts "lam is a class of: #{lam.class.name}"

  puts "lam method count: #{lam.methods.size}"

  puts

  prc = Proc.new { puts 'prc variable assigned a Proc' }

  prc.call

  puts "prc is a class of: #{prc.class.name}"

  puts "prc method count: #{prc.methods.size}"

  ""

end

Этот код напечатает:

— what\_are\_lambdas\_and\_procs —  
lam variable assigned a lambda  
lam is a class of: Proc  
lam method count: 55

prc variable assigned a Proc  
prc is a class of: Proc  
prc method count: 55

Так в чем же различия между Proc и lambda?

Таких отличий всего два:

1. Различие в том, как Proc / Lambda назначает аргументы

2. Различие в том, что происходит, когда Proc / Lambda вызовы возвращают внутри вызываемого метода.

**Присвоение значений аргументам proc и lambda**

Если лямбда вызывается с большим, или меньшим количеством аргументов, чем необходимо, тогда Ruby выдает ошибку **ArgumentError**.

Однако когда Proc вызывается с большим количеством аргументов, чем необходимо, никакой ошибки не возвращается и лишние аргументы просто отбрасываются. Когда процедура вызывается с меньшим количеством аргументов, то те параметры, которые не получили необходимых значений, приобретают значение nil.

Следующие два фрагмента кода демонстрируют вышеописанное различие в работе:

def lambda\_args

  puts "--- #{\_\_method\_\_} ---"

  lam = lambda { |a, b| puts "lambda with 2 arguments a:#{a}, b:#{b}" }

  lam.call( "first", "second")

  puts "lambda called with three arguments..."

  begin

    lam.call("first", "second", "third")

  rescue

    puts "   ArgumentError: wrong number of arguments (3 for 2)"

  end

  puts "lambda called with no arguments..."

  begin

    lam.call()

  rescue

    puts "   ArgumentError: wrong number of arguments (0 for 2)"

  end

end

def proc\_args

  puts "--- #{\_\_method\_\_} ---"

  prc = Proc.new { |a, b| puts "Proc with 2 arguments a:#{(a.nil?) ? "nil" : a }, b:#{(b.nil?) ? "nil" : b }" }

  prc.call( "first", "second")

  puts "Proc called with three arguments..."

  prc.call("first", "second", "third")

  puts "Proc called with no arguments..."

  prc.call()

end

Они намечатают следующий текст:

— lambda\_args —  
lambda with 2 arguments a:first, b:second  
lambda called with three arguments…  
ArgumentError: wrong number of arguments (3 for 2)  
lambda called with no arguments…  
ArgumentError: wrong number of arguments (0 for 2)

— proc\_args —  
Proc with 2 arguments a:first, b:second  
Proc called with three arguments…  
Proc with 2 arguments a:first, b:second  
Proc called with no arguments…  
Proc with 2 arguments a:nil, b:nil

**Возврат значений из proc и lambda**

Когда лямбда-функция возвращает значение и эта лямбда вызывается внутри метода, то метод будет просто продолжать свое выполнение после окончания работы лямбды, пока в конце-концов он сам не сам не вернет свое собственное значение.

Однако, когда Proc возвращает значение и эта процедура вызывается внутри метода, тогда выполнение этого метода будет остановлено. Это немного странное поведение и поэтому Proc следует использовать осторожно.

Два следующих примера демонстрируют это поведение:

def lam\_return

  puts "--- #{\_\_method\_\_} ---"

  puts "simple function that calls a lambda and then returns it's own string."

  my\_lam = lambda { return 'Lambda has returned' }

  my\_lam.call

  return 'lam\_return method string returned'

end

def proc\_return

  puts "--- #{\_\_method\_\_} ---"

  puts "simple function that calls a Proc, \nBUT instead of returning it's own string returns the Proc's string."

  my\_proc = Proc.new { return 'Proc string returned' }

  my\_proc.call # this function returns and finishes here

  return 'proc\_return method string returned' # the return is never called

end

Они напечатают следующее:

— lam\_return —  
simple function that calls a lambda and then returns it’s own string.  
lam\_return method string returned

— proc\_return —  
simple function that calls a Proc,  
BUT instead of returning it’s own string returns the Proc’s string.  
Proc string returned

**Обход ограничений proc на возврат значений**

Выше уже говорилось, что завершение работы Proc автоматически завершает работу метода его вызвавшего, однако существует простой способ обхода этой особенности. Для того, чтобы избавится от этой особенности, следует просто избавится от выражения return в Proc, таким образом Proc не будет завершать работу вызвавшего ее метода, но значение из Proc, тем не менее будет возвращено, поскольку в Ruby все имеет значение и если оно явно не возвращается через выражение return, то возвращается результат результат последней строки. Ниже приведен пример:

def proc\_return\_work\_around

  puts "--- #{\_\_method\_\_} ---"

  puts "simple function that calls a Proc, \nhowever the method string still returns."

  my\_proc = Proc.new { 'Proc string returned' }

  my\_proc.call # this function returns and finishes here

  return "#{\_\_method\_\_} method string returned" # the return is never called

end

Этот код напечатает следующее:

— proc\_return\_work\_around —  
simple function that calls a Proc,  
however the method string still returns.  
proc\_return\_work\_around method string returned

**Заключение**

Как вы заметили, существует только два маленьких отличия между процедурой и лямбда-функцией, но эти отличия имеют весьма интересное поведение. Поэтому следуя принципу «меньшей неожиданности» в Ruby, лучше использовать Lambda вместо процедур, если только использование процедур не является необходимым.