

Занятие: 3

Циклы и управление потоком выполнения:

Вспомнили `if..end` / `case` - тоже управление потоком

Бесконечный цикл:

```
loop do
  end
```

Итерация по коллекции:

```
for element in somewhere
  puts element
end
```

- `for` есть для всех, у кого есть `#each`. Почему? **?**
- "Ruby way" - использовать `#each`

Циклы с условием:

```
while <условие_правдивости> do # do не обязательно
  # не забыть изменять условие правдивости
end
```

Упрощённая форма: `do_something while <истина>`

```
until <ложное_условие_к_которому_стремимся> do # do не обязательно
  # не забыть изменять ложное_условие_к_которому_стремимся
end
```

Упрощённая форма: `do_something until <ложное_условие_к_которому_стремимся>`

Прерывание циклов:

- `break` — выходит из цикла к следующей инструкции
- `redo` — начинает цикл сначала, но без пересчёта условия цикла или следующего элемента (если для итераторов — повтор прохода с текущим значением)
- `next` — пропускает остаток и начинает следующий проход этого же цикла
- `retry` — запускает тело сначала (обнуляет значение) (пока пропускаем, так как оно работает только при наличие обработчика исключений)

Примеры

Вывод всех элементов массива:

```
array = [1, 2, 3, 4, 5]
array.each do |i|
  puts i
end
```

Особая форма: вывод `i` от 5ти до 10:

```
i = 5
while (i == 5)...(i == 10) do
  puts i
  i += 1
end
```

```
end
```

Вывод чисел от 5 (4 + 1) до 10 (не включая), исключив 6:

```
x = 4
while x < 10
  x += 1
  redo if x == 6
  puts x
end
```

(пока меньше 3х перезапускаем, выводим 3..5, дальше срабатывает `break`):

```
i=0
loop do
  i += 1
  next if i < 3
  puts i
  break if i > 4
end
# => 3, 4, 5
```

Дополнительный индекс, разобрать самостоятельно:

```
a = 0 # => 0
i = 0 #=> 0
while a < 10
  puts a
  i += 1
  redo if a == 4 && i < 10
  a += 1
end
```

Интересные трюки и работа с датой:

```
t = Time.now
=> "20150112 23:31:50 +0300"
t.hour
=> 23
t.monday?
=> true
t.utc
=> 20150112 20:31:50 UTC

%w(1 2 3)
%W(1 #{1+1} 3)
=> ['1', '2', '3']

a = 0
a += 1 # a = a + 1
a -= 1 # a = a - 1
a *= 1 # a = a * 1
a /= 2 # a = a / 2
```

Enumerable и итераторы

Модуль `Enumerable` предоставляет очень мощные **методы** для работы с массивами и хэшами, которые **называются итераторами**.

Итератор — это метод, который обходит коллекцию по элементно выполняя с каждым элементом определенную операцию. Если быть совсем честным, то итератор, в большинстве случаев не сам выполняет работу, а передает каждый элемент коллекции в блок кода, где, чаще всего и происходит основная работа.

Давайте рассмотрим основные методы—итераторы:

- `each` (на `Array` и `Hash`)

- `each_with_index` (конвертирует `Hash` в `Array`)
- `collect` / `map`
- `count`
- `any?` / `all?`
- `find` / `detect` (лучше `detect` из-за `Rails AR`)
- `find_all` / `select`
- `each_char` / `chars` - на строках
- `inject`
- ...
- итераторы без блока возвращают объект типа `Enumerator`
- модуль `Enumerable` (он же примесь)
- модуль `Enumerable` подмешан в `Array`, `Hash`, вот откуда у них часть методов

Обзор

`#each` - данный итератор пробегает по коллекции и передает каждый ее элемент в блок кода, где над ними происходит определенное действие:

```
a = [1,2,3,4,5]
h = {}
a.each { |value| h[value] = value ** 2 } #=> [1, 2, 3, 4, 5]
h #=> {1=>1, 2=>4, 3=>9, 4=>16, 5=>25}
(1..5).each{|v| print v**2, " "} #=> 1 4 9 16 25
```

Метод `#each` для хэшей имеет некоторое отличие от аналога для массивов — он может принимать не только значения, но и индексы:

```
h = {a: 100, b:200, c:300, d:400}
h.each { |value| print value, " " }
# =>[:a, 100] [:b, 200] [:c, 300] [:d, 400]
h.each{|key, value| print key, "> ", value, " " }
# => a> 100 b> 200 c> 300 d> 400
```

`#each_with_index` - данный метод предоставляет доступ не только к значениям элементов, но и их индексам:

```
a.each_with_index { |value, key| a[key] = value ** 2 }
# => [1, 4, 9, 16, 25]
```

При использовании `#each_with_index` вместе с диапазонами, порядок следования аргументов блока (ключа и значения) меняется:

```
(1..5).each_with_index { |key, value| print value, "", key, " "}
01 12 23 34 45
```

При выполнении `#each_with_index` для хэша происходит его преобразование в массив:

```
h = {a: 100, b: 200, c: 300, d: 400}
h.each_with_index { |v, k| print "#{k} > #{v} " }
0 -> [:a, 100] 1 -> [:b, 200] 2 -> [:c, 300] 3 0> [:d, 400]
```

`#collect` и `#map` - методы синонимы, которые используются для создания новой коллекции из уже имеющейся:

```
a = [1,2,3,4,5]
a2 = a.collect { |v| v**2 }
a2 #=> [1, 4, 9, 16, 25]
```

`#collect` и `#map` используются для создания коллекций, где корнем является массив:

```
h2 = h.collect { |k, v| { "#{v}#{k}" => v ** 2 } }
# => [{"100a"=>10000}, {"200b"=>40000}, {"300c"=>90000}, {"400d"=>160000}]
```

Для того, чтобы получить хэш без вложенных коллекций следует использовать уже знакомый нам способ создания хеша при помощи `Hash[]` и метод `#flatten`:

```
h2 = Hash[*h.collect { |k, v| ["#{v}#{k}", v**2] }.flatten]
# => {"100a"=>10000, "200b"=>40000, "300c"=>90000, "400d"=>160000}
```

`#count` - подсчёт элементов по заданному критерию. Может быть вызван и без блока:

```
h #=> {:a=>100, :b=>200, :c=>300, :d=>400}
h.count { |k,v| v > 200 } #=> 2
h.count { |k,v| v >= 200 } #=> 3
a #=> [1, 2, 3, 4, 5]

a.count { |v| v % 2 == 0 } #=> 2
(0..10).count { |v| v**2 > 16 } #=> 6
```

`#any?` и `#all?` - данные методы несколько отличаются от остальных итераторов тем, что не возвращают коллекцию, а возвращают булево значение `true` или `false`. Эти методы проходят по элементам коллекции и передают каждый из них в блок кода, который выполняет проверку. Метод `#any?` возвращает `true`, если хотя бы для одного элемента коллекции блок кода возвращает `true`, в противном случае будет возвращено значение `false`, метод `#all?` возвращает `true` только когда для всех элементов коллекции условие в блоке кода возвращает `true`, иначе возвращает `false`:

```
a #=> [1, 2, 3, 4, 5]
a.any? { |v| v > 4 } #=> true
a.all? { |v| v > 4 } #=> false
a.any? { |v| v > 0 } #=> true
a.all? { |v| v > 0 } #=> true
h #=> {:a=>100, :b=>200, :c=>300, :d=>400}

h.all? { |k,v| v > 100 } #=> false
h.any? { |k,v| v > 100 } #=> true
h.any? { |k,v| k.equal? :b } #=> true
(0..100).any? { |v| v > 100 } #=> false
(0..100).all? { |v| v < 100 } #=> true
```

`#find` и `#detect` - данные итераторы являются синонимами. Я рекомендую вам использовать метод `#detect`, поскольку метод `#find` много где переопределен. Методы `#find` и `#detect` используются для получения **первого** элемента коллекции который соответствует условию в блоке кода:

```
a #=> [1, 2, 3, 4, 5]
a.detect { |v| v >= 3 } #=> 3
h #=> {:a=>100, :b=>200, :c=>300, :d=>400}
h.detect { |k,v| v % 2 == 0 } #=> [:a, 100]
h.detect { |k,v| v % 15 == 0 } #=> [:c, 300]
(0..10).detect { |v| v**2 == v } #=> 0
```

`#find_all`, `#select` - данные методы выполняют ту же работу, что и `#find` и `#detect`, однако возвращают не первый элемент, который соответствует условию, а массив всех соответствующих условию элементов:

```
a #=> [1, 2, 3, 4, 5]
a.find_all { |v| v >= 3 } #=> [3, 4, 5]
h #=> {:a=>100, :b=>200, :c=>300, :d=>400}
h.select { |k,v| k.instance_of?(Symbol) } #=> [:a=>100, :b=>200, :c=>300, :d=>400]
(0..100).select { |v| v > 30 and v < 40 } #=> [31, 32, 33, 34, 35, 36, 37, 38, 39]
```

`#each_char` и `#chars` - данные итераторы, как не странно, принадлежат строковым объектам и передают при каждой итерации по одному символу в блок кода:

```
str = ""
```

```
"hello readers!".chars { |char| str << char.upcase}
str #=> "HELLO READERS!"
```

Тема: Методы, блоки, `yield`, `return`, `lambda`, `proc`

Блоки-Проки-Лямбды

- почти как методы (методы есть их дети)

`... {}` - блок `... do..end` - блок

```
Proc.new do [| params| ]
  <code>
end # уже объект, `Proc`
```

Вызов происходит через `#call` или `#[]`.

```
greater = Proc.new do
  puts "Hello, Ruby!"
end

puts greater.class
greater.call

greater2 = Proc.new do |word|
  puts word
end
greater2.call "Hello, Ruby!"
```

Блоки – недоПРОКИ, жалкие синтаксические структуры 😊 Помимо этого, как мы увидим совсем скоро, методы могут принимать только один блок, но много Проков.

`#lambda` – метод модуля `Kernel`, считай глобальный метод.

Зачем, если есть методы?

- методы не могут быть присвоены, так как не объекты
- они связаны с объектом, для которого определены. (вспоминаем `self`)
- все глобальные методы – на самом деле на `Object` и `Kernel`
- объект, обладатель метода – получатель сообщений, получая их(сообщения) он выполняет инструкции методов

В `Ruby` даже есть `#send` на `Object`

- а раз методы не объекты – то не могут получать сообщения

Разница между `Proc` и `lambda`:

- `lambda` проверяет количество аргументов, `proc` - нет
- работа с `return` (разберём дальше)

```
def double_trouble
  yield
  yield
end
double_trouble {puts "Hi!"}
```

```
def double_trouble2(block)
  block.call
  block.call
end
double_trouble2 lambda {puts "Yo!"}
```

Как методы вызывают Проки? => `yield` !

Методы

- не живут сами по себе, связаны с объектом
- не могут принимать другие методы равно как и возвращать (только вызывать) — но могут делать это с `Proc`
- создание через вызов `def` происходит в контексте объекта, в котором мы находимся (в `irb` - это `main`)
- вызов — по имени. Скобки не обязательны

```
def method_name
  result = something
  return result
end
```

```
(variable = method_name) == (variable = method_name())
```

- выводят последнее выполненное выражение, `return` — в большинстве случаев не нужен.

Инициализация

```
def method_name
  result = something
end
```

Принятие аргументов

```
def method_name(param1, param2)
  puts param1
end
```

```
method_name(1, 'two') <==> method_name 1, 'two'
```

- не обязательные аргументы (по-умолчанию)

```
def method_name(param1, param2 = 3, param3)
  <code>
end
method_name(1, 2) # => 2 присвоится param3
method_name(1, 3, 4)
method_name(1)
=> ArgumentError: wrong number of arguments (1 for 2..3)
```

- а вообще количество параметров в определении и вызове должно совпадать, мы же помним почему? 😊

Говорящие имена методов:

```
method_name!
method_name?
method_name= # => writer
```

- сворачивание аргументов:

```
def method_name(a, *b)
  end
# теперь он гибок как гимнаст :)
method_name(1), method_name(1,2,3,4)

def method_name(*a)
  <code>
end
method_name, method_name(1), method_name(1,2,3,4)
```

Методы и Блоки:

```
def method_name
  yield
end
```

```
end
method_name {puts "Something"} # => блок, не Proc. Почему?
```

`#block_given?` - проверяет, был ли передан блок методу
`yield` быстрее чем `#call`

- передача блока как параметра:

```
def method_name(&block_body)
  block_body.call
end
```

- должен быть последним

```
def method_name(a, b, &block_body)
  block_body.call(a + b)
end
```

- что будет если добавить ещё и `yield` ?
- как происходит обработка параметров, если блок их не принимает?
- методы могут возвращать `Proc` 's!
- переменные методов – локальные(доберёмся ещё до области видимости)

`Object#respond_to?` / `Module#method_defined?` - проверка, есть ли у объекта данный метод

Keyword arguments:

```
def foo(str: "foo", num: 424242)
  [str, num]
end

foo(num: 10, str: "string")
# порядок не важен
# как было раньше?
```

Преобразование символа в процедуру:

Ruby является не просто объектно-ориентированным языком программирования, но поддерживает и другие парадигмы, например функциональную. Таким образом имеет в своем арсенале процедуры, которые являются объектами. У символов есть очень мощный и полезный метод `#to_proc`, который позволяет создать из символа процедуру. Созданная таким способом процедура является ни чем иным, как вызовом в блоке кода метода соответствующего символу:

```
cap = :capitalize.to_proc #=> #<Proc:0x9624040>
cap.call("hello world") #=> "Hello world"
proc = :blablabla.to_proc #=> #<Proc:0x962bcd>
proc.call("hello!") #=> NoMethodError: undefined method `blablabla' for "hello!":String
```

`#to_proc` - просто создает процедуру с вызовом одноименного с символом метода, в первом случае эта процедура будет иметь вид: `proc { |arg| arg.capitalize }`, а если быть совсем точным, то `proc { |arg| arg.send(:capitalize) }`. Мы уже знаем, что `#call` используется для вызова процедуры (ее выполнения), а `#send` для вызова соответствующего символу метода у объекта — приемника, поэтому `"hello".send(:upcase)` равносильно `"hello".upcase`.

Когда это может пригодиться? В блоках и лямбдах!

```
%w(a b c d e).map(&:upcase)
=> ["A", "B", "C", "D", "E"]
# Класс!
```

Вернёмся к прокам и лямбдам.

Разница между `proc` и `lambda`:

- `lambda` проверяет количество аргументов, `proc` - нет
- `return` из `proc` выходит из метода его включающего
- `return` из `lambda` выходит из вызова самой лямбды, но не метода

```
def try_proc
  our_proc = Proc.new { return "EXIT" }
  our_proc.call
  "Что хочу то и пишу"
end
puts try_proc
EXIT
=> nil
```

```
def try_lambda
  our_lambda = lambda { return "EXIT" }
  our_lambda.call
  "Тут уж нужно осторожно"
end
puts try_lambda
Тут уж нужно осторожно
=> nil
```

Ещё немного о методах:

Именованные, не обязательные аргументы задаются через `**имя` и аккумулируются в переменную с этим именем, равную хэшу:

```
def foo(str: "foo", num: 424242, **options)
  puts [str, num].inspect
  puts options.inspect
end

foo(str: 1, num: 2, test: 1, test2: 2)
[1, 2]
{:test=>1, :test2=>2}
=> nil
```

Полная возможная форма:

```
def my_method(first_value, *values, name: 'default', **ignore_extra, &block)
  puts first_value
  puts values.inspect
  puts name
  puts ignore_extra.inspect
end

my_method(1)
1
[]
default
{}
=> nil

my_method(1, 2, 3, 4)
1
[2, 3, 4]
default
{}
=> nil

my_method(1, 2, name: "test", ruby: "cool")
1
[2]
test
{:ruby=>"cool"}
=> nil
```

Алиасы:


```
alias :new :old # ключевое слово, определяется в момент считки кода
alias_method :new, :old # метод на Module, определяется во время выполнения
```

```
def test
  puts '1'
end
alias :new_test :test
new_test # => '1'
```

- `undef :new_test` # удаляет существующий метод
- звездочка в вызове чуть другое нежели в описании метода:

```
def method_name(a, b, c)
  <code>
end

method_name(*[1,2,3]) # передача для ленивых :)
```

Когда использовать return в методах?

Мы помним, что для методов не нужно явно указывать возвращаемое значение, они вернут результат последнего выполнения, 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 устывает то, что метод возвращает результат выполнения последней строки.

Есть у блока с методом еще одно общее свойство: оба они создают новую локальную область видимости переменных. Другими словами, переменная, объявленная внутри них, недоступна в других участках выполняемого кода. Примеры:

```
def test
  x = 1
end
=> :test
x
NameError: undefined local variable or method `x' for main:Object

test # => 1
x
NameError: undefined local variable or method `x' for main:Object
```

Как мы видим, переменная `x`, не появилась, потому что она является локальной для метода, и не попадает в общую область видимости.

Аналогично для блоков:

```
p = proc { |x| x = 1 }
=> #<Proc:0x000000022b4268@(irb):46>
x
NameError: undefined local variable or method `x' for main:Object

p.call(1) # => 1
x
NameError: undefined local variable or method `x' for main:Object
```

Отличия Методов от Блоков:

Метод всегда возвращает значение: это или значение последнего выражения, или значение выражения после ключевого слова `return`. Блок тоже возвращает значение, и его можно получить в методе как результат вызова `yield`.

Блоки видят переменные объявленные в области действия блока (работают как замыкания), а методы нет.

Пример с использованием блоков:

```
hello = "Hello World"
l = lambda { puts hello }
```

```
1.call #=> Hello World => nil
```

Пример с использованием метода:

```
def say_hello
  puts hello
end
hello = "Hello World"
say_hello #=> NameError: undefined local variable or method `hello'

cat = "Голодный кот"
=> "Голодный кот"
def locate_cat
  puts "Кажется, #{cat} на крыше."
end
=> nil

locate_cat
NameError: undefined local variable or method 'cat' for main:Object

def whatever
  yield
end
=> nil
whatever do
  puts "#{cat} добрался до колбасы!"
  cat = "Уже не голодный кот"
end

Голодный кот добрался до колбасы!
=> "Уже не голодный кот"
puts cat
Уже не голодный кот
=> nil
```

Столкновение переменных

Что произойдет, если имя аргумента блока совпадает с внешней переменной? В этом случае Ruby создаст новую локальную переменную, и внешняя переменная внутри блока будет недоступна.

```
def with_one
  yield(1)
end
number = 99
with_one { |number| puts "number равно #{number}" }

number равно 1
=> nil
puts number
99
=> nil
```

Наконец, а что делать, если вы хотите, чтобы блок **не** захватывал внутри и менял внешнюю переменную? Для этого нужно в принудительном порядке создать одноименную локальную переменную блока, делается это ее указанием после точки с запятой в списке аргументов:

```
puts number
99
=> nil

with_one do |i; number|
  puts "i равно #{i}"
  number = 123
end
i равно 1
=> 123

puts number
```

```
99
=> nil
```

`with_one` попрежнему передает в блок единицу.

Как это использовать?

Поначалу может показаться, что всё это жадное «захватывание» переменных блоком (поэтому блоки еще называют "замыканиями") не имеет никакого смысла, если не хуже — представляет опасность: можно очень легко изменить переменную, полагая, что работаешь с ней только внутри блока.

На самом деле, практическое применение замыканий становится понятным, только когда их начинают *перемещать между областями видимости*, что выходит за рамки нашего курса. Поэтому пока что стоит принять на веру, что замыкание — полезная штука, если работать с ней аккуратно. 😊

Блок и хеш

Так получилось, что у блока и хеша одинаковая нотация: фигурные скобки. Поэтому, если попытаться передать последний в качестве аргумента, не взяв в круглые скобки, `Ruby` примет его за блок, и выдаст ошибку. Если сохранять стиль, нужно опустить и фигурные скобки:

```
kick_ass { :joker => "2 паза", :penguin => "1 паз" } # синтаксическая ошибка
kick_ass :joker => "2 паза", :penguin => "1 паз"    # так всё нормально
```

Наконец, если вам нужно передать методу и хеш, и блок, или используйте исключительно `do .. end`, или забудьте о поэтичном стиле и берите аргументы в скобки, как положено.

Bonus:

Новый с `Ruby 1.9.1`, синтаксис лямбды: Было:

```
c = lambda { |a, b| [b, a] }
c.call(1, 2)

# Стало:
c = ->(a, b) { [b, a] }
c.call(1, 2)
```

Домашнее задание: 3

Теория:

- прочесть заметки лекции ещё раз, два, три...
- Осмотреться в документации:
 - <http://ruby-doc.org/core-2.2.0/Enumerable.html>
 - <http://www.ruby-doc.org/core-2.2.0/Proc.html>
- Почитать про Блоки и Лямбды:
 - <http://nashbridges.me/procs-and-lambdas>
- Почитать про методы и замыкания (пункты 1.1 и 1.2):
 - https://ru.wikibooks.org/wiki/Ruby/%D0%9F%D0%BE%D0%B4%D1%80%D0%BE%D0%B1%D0%BD%D0%B5%D0%B5_%D0%BE_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B0%D1%85
- ** прочитайте главы о `Enumerable` и итераторов, проках, лямбдах и методах в любой из книг по `Ruby`
- составить список вопросов
- выделить одну интересную и запомнившуюся особенность/метод/факт связанный с `Ruby`

Практика:

- написать метод, который принимает аргументом объект типа `Proc` и возвращает время затраченное на выполнение данного прока (примитивный бенчмаркинг).
Текущее время можно узнать с помощью `Time.now`
- дан массив: `array = [1, 2, 3, 4]`. Написать метод `mega_sum`, который поддерживает следующее использование:

```
o mega_sum(array) # => 10 # простая сумма элементов
o mega_sum(array, 10) # => 20 # сумма всех элементов + переданное значение
o mega_sum(array) { |i| i ** 2 } => 30 # сумма элементов с применённым блоком
o mega_sum(array, 10) { |i| i ** 2 } => 40 # сумма элементов с применённым блоком + переданное значение Это всё
  один метод!
```

- В диапазоне от 1 до 1_000, найти первое трёхзначное число у которого остаток деления на 7 равен 3
- Написать метод, с двумя именованными аргументами, а всеми остальными - не-обязательными (не ограничено количеством). Если за методом следует блок, то вызвать его на каждом из не обязательных аргументов, если же блок не следует вывести 'ERROR'. Пример:

```
method_name(...) # => "ERROR"
method_name(..., 1, 2, 3) { |i| puts i }
1
2
3
=> nil
```

- Апгрейд стандартной библиотеки. Мы напишем метод, который будет возвращать все чётные числа, соответствующие переданному в блок условию:

Чтобы это работало, делаем так:

```
class Array
  def ваш_метод
    < code >
  end
end
```

```
puts [1, 2, 3, 4, 5, 6, 7].ваш_метод { |i| i > 2 }.inspect # => [4, 6]
puts [1, 2, 3, 4, 5, 6, 7].ваш_метод { |i| i > 10 }.inspect # => nil
puts [2, 4, 6, 8, 10, 12, 7].ваш_метод { |i| i.between?(6, 12) }.inspect # => [6, 8, 10, 12]
```

Домашку присылать по адресу: `aliaksandr_buhayeu@epam.com` с темой письма: `MTN:L_3:NAME_SURNAME` .

На этом всё, жду вас на следующем занятии!