# Типы данных: Hash, Array, Integer, BigInt, Enumerable, …. Итераторы. Оператор ``

### ****Строки****

Строки — это произвольной длины наборы символов, которые заключены в одинарные или двойные кавычки. Пример строк:

“this is a string”  
‘this is a string in single quotes’

Вы также можете заключять строковый литерал в любую другую пару символов, кроме одинарных и двойных кавычек указывая ставя перд ним специальный идентификатор строки %q, например:

|  |
| --- |
| %q!this is a string in single quotes!  %q{this is a string in single quotes} |

Вы не можете использовать только цифры и буквы, все остальные символы доступны для заключения в них строкового литерала. Представленное в примере выше заключение строки в произвольные символы равносильно заключению в одинарные кавычки. Это важно знать потому, что между строкой в одинарных кавычках и в двойных имеется большая разница, но об этом чуточку ниже.  
  
Ruby также позволяет создавать строковые литералы, которые занимают несколько физических строк:

str = "Welcome to

Ruby!"

В результате такого объявления переменная str будет хранить следущую строку: «Welcome to\nRuby blog!», где \n является специальным символом новой строки.

Существует также следующий стиль (более аккуратный и желаемый) для создания многострочных строковых литералов:

str = <<STR

This is a

multiline

string

STR

Здесь, вместо STR — идентификатора конца строки может быть использован любой другой слово, например error\_message, wellcome\_message и так далее.

***Интерполяция строки***  
Наконец-то мы дошли до того момента, когда вы узнаете различие между строками заключенные в одинарные и двойные кавычки. Строки заключенные в двойные кавычки могут содержать выполняемые фрагменты кода, а также специальные символы, например тот же символ новой строки. Помещенный в строку код будет выполнен и на его место в строке будет вставлено возвращенное им значение. Код, который вы желаете поместить в строку следует заключить в фикурные скобки перед которыми должен следовать символ шарп — #{здесь ваш код}, примеры:

a = "String a has #{a.class} type" #=> "String a has String type"

a = 1

b = 2

c = "a + b = #{a + b}" #=> "a + b = 3"

Если вы не желаете, чтобы что-то помещенное в #{} выполнялось как код и чтобы специальные символы содержащиеся в строке игнорировались, вам следует либо использовать одинарные кавычки для заключения в них строки, либо экранировать те фрагменты строки, заключенной в двойные кавычки, которые могут выполняться. Экранирование происходит при помощи символа обратного слэша «\», примеры:

c = 'a + b = #{a + b}' #=> "a + b = \#{a + b}"

c = "a + b = \#{a + b}" #=> "a + b = \#{a + b}"

Обратите внимание, что не зависимо от того, в двойные или одинарные кавычки мы заключаем строку, переменной присваивается строка заключенная в двойные кавычки. Это означает, что в строке заключенное в одинарные кавычки автоматически происходит экранирование.

Не смотря на то, что в строках заключенных в одиночные кавычки экранирование происходит автоматически, все-таки существует один символ, который вам однозначно придется экранировать — это собственно одинарные кавычки, пример:

str = 'hello 'Chuck'!' # Произойдет ошибка!

str = 'hello \'Chuck\'!' #=> "hello 'Chuck'!"

Такое экранирование в одинарных строках необходимо для того, чтобы интерпретатор при чтении кода не путался в том, является ли одиночная кавычка частью строки, или она завершает ее. Именно для таких случаев и была добавлена возможность создания строк при помощи идентификатора строки %q и произвольных обрамляющих символов, так как строка созданная в такой способ абсолютно идентична строке заключенной в одинарные кавычки с той лишь разницей, что вам не нужно экранировать имеющиеся в строке кавычки, пример:

str = %q<Hello 'Chuck'!> #=> "Hello 'Chuck'!"

У способа создания многострочной строки при помощи <<END\_IDENTIFIER имеет также некоторый нюанс, например такой метод создания строк может быть аналогичен как созданию строк в двойных кавычках, так и в одинарных, пример:

str = <<END

#{"Hello Chuck!"}

END

#=> "Hello Chuck!\n"

str = <<"END"

#{"Hello Chuck!"}

END

#=> "Hello Chuck!\n"

str = <<'END' #обратите внимание на кавычки в которые заключен END

#{"Hello Chuck!"}

END

#=> "\#{\"Hello Chuck!\"}\n"

На этом с созданием строк мы покончили, перейдем к такому важному занятию, как работа со строками!

***Конкантенация строк***

Ruby очень изящный язык программирования, который имеет очень удобный и понятный синтаксис и предоставляет очень много встроенных методов для работы с основными типами.

Объединение строк (конкантенация строк). Для объединения строк достаточно использовать оператор + (как не странно он же мето), пример:

str = "first string" + " " + "second string" #=> "first string second string"

Согласитесь, + здесь выглядит куда логичней, чем . в PHP, да и код более читабелен.

Еще один способ объединить строки — это оператор \*, который занимается тем, что объединяет строку со своими копиями N-число раз, пример:

"string " \* 3 #=> "string string string "

***Индексация строк, элементы строк***

В Ruby строки имеют несколько сходств с массивами по той причине, что каждый символ строки имеет свой индекс — порядок в строке по которому к нему можно обратится. Если говорить еще более точно, то таких индекцов целых два: левосторонний и правосторонний, примеры:

str = "abcdefg"

str[0] #=> "a"

str[1] #=> "b"

str[2] #=> "c"

str[-1] #=> "g"

str[-2] #=> "f"

str[-3] #=> "e"

Левосторонние индексы указывают позицию символа с левой стороны (с начала) строки и начинаются с нуля. Правосторонние индексы указывают позицию символа с правой стороны (с конца строки) и начинаются с -1, при этом все правосторонние индексы имеют в себе знак «-», который и отличает их от левосторонних. Терминология «левая сторона строки» и «правая сторона строки» более корректна, поскольку, например, в арабском языке письмо идет справа на лево.

Ruby позволяет вам обращаться не только к отдельному символу в строке, но и к набору таких символов, для этого в квадратных скобках необходимо указывать, либо индекс первого выбираемого символа и количество выбираемых символов, либо диапазон индексов выбираемых символов, примеры:

str[0,str.size] #=> "abcdefg"

str[0,3] #=> "abc"

str[-1,5] #=> "g"

str[0..str.size-1] #=> "abcdefg"

str[0..2] #=> "abc"

str[0..-1] #=> "abcdefg"

str[0..-3] #=> "abcde"

str[0...-3] #=> "abcd"

Вы не только можете получать фрагменты строки, но и устанавливать для них значения:

str #=> "abcdefg"

str[0] = "z"

str #=> "zbcdefg"

str[1..str.size] = "hello"

str #=> "zhello"

str[0..str.size] = ""

str #=> ""

str[0..1000] = "Hello Tom!"

str #=> "Hello Tom!"

***Работа со строками***  
В Ruby практически все за редким исключением является объектом и строки не исключение. Строковый объект носит тип String, то есть является экземпляром класса String. Это означает, что строка, как и любой другой объект имеет соственные методы, некоторые из которых вы уже видели в примерах выше. Чтобы убедиться в том, является ли что-то объектом, достаточно попытаться вызвать у этого «чего-то» метод #class. Метод #class не возвращает сообщение «Эта штуковина — объект», он возвращает класс (тип), которому принадлежит объект. Если что-то не является объектом, то и не имеет методов вообще.

"abcdefg".class #=> String - убеждаемся в том, что строка - объект типа String

Строки имеют несколько методов позволяющих работать с регистром символов:

**#upcase** — данный метод переводит все символы строки в верхний регистр (делает все буквы заглавными), пример:

str = "abcdefg"

str.upcase #=> "ABCDEFG"

str #=> "abcdefg"

Обратите внимание на то, что строка не изменилась, метод #upcase просто возвратил новую строку. Для того, чтобы изменить саму строку для которой был вызван метод нужно использовать BANG-методы. BANG-методы, это такие методы, которые изменяют сам объект, для которого они вызываются. В них нет ничего особого, кроме имени. Для BANG — методов желательно, но не обязательно давать имена, который заканчиваются на «!», в нашем случае это метод #upcase!:

|  |
| --- |
| str.upcase!  str #=> "ABCDEFG" |

**#downcase** — этот метод противоположен методы #upcase, он переводит символы строки в нижний регистр, пример:

str.downcase #=> "abcdefg"

str #=> "ABCDEFG"

str.downcase! #=> "abcdefg"

str #=> "abcdefg"

**#capitalize** — данный метод используется для того, чтобы сделать первую букву строки заглавной:

|  |
| --- |
| str.capitalize #=> "Abcdefg"  str #=> "abcdefg"    str.capitalize! #=> "Abcdefg" |

**#swapcase** — данный метод преобразует регистр символов в противоположный, то есть, заглавные буквы в строчные, а строчные в заглавные, пример:

rb = "RuBy "

rb.swapcase #=> "rUbY"

rb #=> "RuBy"

rb.swapcase! #=> "rUbY"

***Удаление пробелов***  
**#lstrip** и **#rstrip** — данные методы используются для удаления пробелов соответственно с левой строны строки и с правой стороны строки, пример:

str = " abcdefg "

str.lstrip #=> "abcdefg "

str.rstrip #=> " abcdefg"

str.lstrip!.rstrip! #=> "abcdefg"

**#strip** — данный метод объединяет в себе оба #lstrip и #rstrip:

|  |
| --- |
| str = " abcdefg "  str.strip #=> "abcdefg"  str.strip! #=> "abcdefg" |

**#reverse** — данный метод меняет порядок символов в строке на обратный, пример:

str.reverse #=> «gfedcba»  
str.reverse! #=> «gfedcba»  
str.reverse! #=> «abcdefg»

**#size** и **#length** — оба эти метода выполняют одно и то же действие — они возвращают количество символов в строке:

"abcdefg".size #=> 7

"abc".length #=> 3

**#delete** — данный метод удаляет фрагмент строки, пример:

|  |
| --- |
| str #=> "abcdefg"  str.delete("abc") #=> "defg"  str.delete!("abc") #=> "defg" |

**#scan** — метод принимает в качестве аргумента регулярное выражение и возвращает  массив подстрок (фрагментов строки), которые соответствует регулярному выражению. Регулярным выражениям будет посвящена отдельная глава, а пока пример:

|  |
| --- |
| str #=> "defg"  str.scan(/d|e/) #=> ["d", "e"]  str.scan(/de/) #=> ["de"]  str.scan(/[^d,f]/) #=> ["e", "g"] |

**#chomp** — данный метод удаляет окончание строки. В качестве примера хочу привести маленький и глупый метод, который переводит множественную форму английского слова в одиночную форму просто путем удаления окончания:

"fishes".chomp("es") #=> "fish"

"fishes".chomp!("es") #=> "fish"

def to\_singular (word)

  word[-2..-1] == "es" ? word.chomp("es") : word.chomp("s")

end

str = "rabbits"

to\_singular(str) #=> "rabbit"

**#index** и **#rindex** — данные методы принимают в качестве аргумента строку или шаблон регулярного выражения и возвращают соответственно индекс символа строки с которого начинается соответственно первое или последнее соответствие шаблону, примеры:

str = "mama mila ramu"

str.index("m") #=> 0

str.rindex("m") #=> 12

str.index(/ma/) #=> 0

str.rindex(/ma/) #=> 2

**#insert** — данный метод принимает в качестве аргументов строку и позицию вставки (индекс) для вставки данной строки в строку для которой он был вызван, при этом символы находящиеся в том месте, куда производится запись не перезаписываются а сдвигаются вперед, пример:

|  |
| --- |
| str = "hell" #=> "hell"  str.insert(str.size, "o") #=> "hello" |

Метод **#insert** производит изменения непосредственно в самой строке.

**#chars** и **#each\_char** — являются методами итераторами, которые проходят по каждому символу строки, передают его в блок кода, который выполняет над каждым символом определенное действие, пример:

"ruby".chars do |ch|

if i == 0 or i%2 == 0

print ch.upcase

else

print ch

end

i += 1

end

#=> RuBy

**#empty?** — данный метод возвращает значение true если строка пустая и false, если в строке имеется хотя бы один символ:

|  |
| --- |
| "".empty? #=> true  "hello".empty? #=> false |

**#include?** — метод принимает фрагмент строки и проверяет строку для которой был вызван на то, существуют ли в ней подстроки соответствующие фрагменту, пример:

|  |
| --- |
| "hello".include?("ll") #=> true  "hello".include?("bye") #=> false |

**#concat**, **+=** и **<<** — все три метода выполняют одно и то же действие — добавляют к существующей строке строку, что была передана в качестве аргумента, примеры:

|  |
| --- |
| str = "ruby"  str << " on rails"  str #=> "ruby on rails"    str = "ruby"  str += "on rails"  str #=> "ruby on rails"    str = "ruby"  str.concat("ruby")  str #=> "rubyruby" |

### ****Числа****

Число, так же, как и строка является объектом в Ruby. Числовой объект в отличие от строкового может иметь целых три типа: **Fixnum**, **Bignum** и **Float**. Fixnum и Bignum используются для представления целых чисел и практически идентичны за исключение того, что Bignum используется для хранения более больших чисел, чем может хранить Fixnum и содержит несколько дополнительных методов. Оба класса Bignum и Fixnum наследуются от класса Integer, который в свою очередь наследуется от класса Numeric — прародителя всего, что связано с числами в Ruby. Объектом типа Float является число с плавающей точкой, то есть дробное число, доказательства:

5.class #=> Fixnum

999999999999999999.class #=> Bignum

5.6.class #=> Float

5.6.class.superclass #=> Numeric

5.class.superclass #=> Integer

999999999999999999.class.superclass #=> Integer

Integer.superclass #=> Numeric Удобное представление чисел, при помещение в код больших чисел, вы можете использовать разделители разрядом для улучшения читабельности кода, например:

100\_000\_000 #=> 100000000

В Ruby можно с одинаковой легкостью работать сразу в нескольких системах исчислений. Для представления числа в каждой системе исчисления имеется специальный формат записи числа:

Бинарного (binary numbers):

|  |
| --- |
| 0b1 #=> 1,  0b10 #=> 2  0b100 #=> 4 |

Восьмиричного (octal numbers):

0o1 #=> 1

0o10 #=> 8

0o100 #=> 64

Шестнадцетиричного (hexadicimal numbers):

0x1 #=> 1

0xf => 15

0x10 #=> 16

0x100 #=> 256

***Работа с числами***

***Арифметические операции***  
Все арифметические операции реализованы в виде методов, однако, не смотря на это, как операторы имеют приоритет выполнения, например у метода выполняющего умножение приоритет больше, чем у метода выполняющего сложение, как и в математике. Примеры работы с числами и краткое пояснение:

**+**, **-**, **\***, **/**, **\*\*** — методы нахождения суммы, разности, произведения, результата деления, и возведения в степень, примеры:

c = 1 + 2 + 3 #=> 6

1 \* 3 + 5 / 5 #=> 4

10 \*\* 2 #=> 100

5 \*\* 3 #=> 125

**#remainder** и **%** — данные методы возвращают остаток от деления, примеры:

10 % 3.5 #=> 3.0

10.remainder 3.5 #=> 3.0

**#div**, **/** и **#fdiv** — данные методы возвращают соответственно результат деления нацело(/ и #div) и полный результат, включающий дробную часть (#fdiv), пример:

3.div 4 #=> 0

3 / 4 #=> 0

3.fdiv 4 #=> 0.75

10.div(3.5) #=> 2

10 / 3.5 #=> 2.857142857142857

10.fdiv 3.5 #=> 2.857142857142857

Метод **/** отличается от метода #div тем, что возвращает результат в зависимости от типа делимого и делителя. Если один из членов выражения — дробное число, то и результат деления будет представлен дробным числом, то есть без остатка. #div всегда возвращает результат деления нацело, то есть без дробной части.

**#divmod** — данный метод совмещает в себе функции методов #div и %, то есть возвращает массив состоящий из двух элементов — результата деления на цело и остатка, пример:

11.divmod 4 #=> [2, 3]

11.divmod 3.3 #=> [3, 1.1000000000000005]

**#next**, **#succ** — оба метода выполнятю одно и то же, а именно: возвращают число болешее на единицу (только для целых чисел), пример:

1.next #=> 2

1.next.next.next #=> 4

5.succ #=> 6

**#abs** и **#magnitude** возвращают модуль числа, пример:

|  |
| --- |
| -10.abs #=> 10 |

**#odd?** и **#even?** — методы возвращают true, если число соответственно нечетное и четное и false, если наоборот, примеры:

1.even? #=> false

1.odd? #=> true

2.even? #=> true

2.odd? #=> false

***Сравнение чисел***

1 <= 2 #=> true -Меньше или равно?

1 < 2 #=> true -Больше?

1 == 1 #=> true -Равно ли?

1 > 3 #=> false -Больше?

1 <= 1 #=> true -Меньше или равно?

Метод **<=>** используется для сравнения двух чисел a и b и возвращает 0, 1, -1 — соответственно, если a == b, если a > b и если a < b, примеры:

|  |
| --- |
| 1 <=> 1 #=> 0  1 <=> 2 #=> -1  1 <=> -1 #=> 1 |

**#zero?** — метод проверяет является ли число нулем, если да, то возвращает true, иначе false, пример:

0.zero? #=> true

4.zero? #=> false

**#nonzero?** — данный метод возвращает само число, если оно не является нулем, если че число — ноль, то возвращает значение nil, что оценивается как false, пример:

0.nonzero? #=> nil

4.nonzero? #=> 4

***Преобразование***

**#to\_f** — метод используется для преобразования целого числа в число с плавающей точкой. Если говорить более корректно, то целочесленное представление заменяется на представление с плавающей точкой, пример:

2.to\_f #=> 2.0

2 / 3 #=> 0

2 / 3.to\_f #=> 0.6666666666666666

**#to\_s** — метод преобразует число в строку, пример:

125.to\_s #=> "125"

**#to\_s** позволяет задавать систему исчисления в котором будет представлено числа, а точнее строковый аналог числа (только для целых чисел), пример:

# двоичное представление

1.to\_s(2) #=> "1"

2.to\_s(2) #=> "10"

889.to\_s(2) #=> "1101111001"

# шестнадцатиричное представление

8.to\_s(16) #=> "8"

9.to\_s(16) #=> "9"

23.to\_s(16) #=> "17"

998.to\_s(16) #=> "3e6"

# восьмиричное представление

8.to\_s(8) #=> "10"

12.to\_s(8) #=> "14"

#любое, какое захотите!

1.to\_s(12) #=> "1"

13.to\_s(12) #=> "11"

**#to\_i** и **#to\_int** приводят объект в целочисленный тип, пример:

1.to\_i #=> 1

1.1.to\_i #=> 1

5.9.to\_i #=> 5

5.9.to\_int #=> 5

"20 dollars per hour".to\_i #=> 20

**#to\_f** — приводит целое число или строку в число с правающей точкой, пример:

"20 dollars per hour".to\_f #=> 20.0

2.to\_f #=> 2.0

***Округление***  
В Ruby существует целых три метода для округления числа: **#ceil**, **#floor** и **#round**. #round занимается тем, что по нормальным правилам округляет число до заданой степени точности, по умолчанию до целого, однако в качестве аргумента ему можно передать разряд, до которого следует округлить число. Метод #floor округляет число всегда до ближайшего меньшего целого числа, а #ceil, наоборот, до ближайшего большего целого, примеры:

1.978.round #=> 2

1.978.round(2) #=> 1.98

1.978.ceil #=> 2

1.978.floor #=> 1

***Целочисленные итераторы***

**#times** — выполняет блок кода N-число раз, где N — число для которого был вызван данный метод:

3.times {|i| print i, ", "} #0, 1, 2,  => 3

**#step** — данный очень похож на метод #times с тем отличием, что имеется возможность задавать шаг хода, например:

# начиная с 1 продолжая до 20 с шагом 2 выполнять блок кода:

1.step(20,2){|x| print x,", "} #1, 3, 5, 7, 9, 11, 13, 15, 17, 19,  => 1

# начиная с 10 продолжая до 20 с шагом 2 выполнять блок кода:

10.step(20,2){|x| print x,", "} #10, 12, 14, 16, 18, 20,  => 10

### ****Диапазоны****

Диапазоны — это особый тип данных в Ruby, который является достаточно уникальным, поскольку далеко не в каждом яыке он существует и достаточно полезным, поскольку позволяет, порой, значительно упростить код.

Диапазон заключается в круглые скобки, и состоит из минимума и максимум, которые разделены двумя или тремя точками, ниже приведен простые примеры диапазонов:

(1..10) # соответствует последовательности от 1 до 10 включительно

(1...10) # соответствует последовательности от 1 до 10, не включая 10

('a'..'z') # соответствует всему латинскому алфавиту.

Диапазоны не богаты на методы и методы для диапазонов используются достаточно редко, за исключением метода to\_a, который преобразует диапазон в массив.  
**Работа с диапазонами**  
**#to\_a** преобразует диапазон в массив, пример:

(1..3).to\_a #=> [1, 2, 3]

|  |
| --- |
|  |

**#min** и **#first**, **#max** и **#last** — данные методы возвращают соответственно начало диапазона и его конец, пример:

(1...100).min #=> 1

(1...100).max #=> 99

(1..100).max #=> 100

(1..100).first #=> 1

(1..100).last #=> 100

('a'..'z').min #=> "a"

('a'..'z').max #=> "z"

Между #min и #first, а также #max и #last существует маленькое, но очень важное различие, незнание которого может привести к ошибкам в коде! Это различие проявляется тогда, когда используются обратные диапазоны, пример:

(1..100).first #=> 1

(1..100).min #=> 1

(100..1).min #=> nil

(100..1).first #=> 100

(1..100).max #=> 100

(1..100).last #=> 100

(100..1).max #=> nil

(100..1).last #=> 1

Как видите, #min и #max при обратных диапазонах возвращают значение nil, этим можно пользоваться, когда вам необходимо узнать, диапазон прямой или обратный, во всех остальных случаях я бы рекомендовал использовать методы #last и #first, если разумеется нет ограничения на то, каким должен быть диапазон (обязательно ли он должен быть прямым?).

**#include?** и **#member?** — методы синонимы, которые принимают в качестве аргумента число или буквенный символ и проверяют его вхождение в диапазон. Если буква или число входит в диапазон, то будет возвращено значение true, иначе false, пример:

(1..100000).include?(1) #=> true

(1..100000).include?(100001) #=> false

(1..100000).member?(100001) #=> false

('a'..'y').member?("c") #=> true

('a'..'y').member?("z") #=> false

**#begin** — метод идентичен методам #first и #min и возвращает первый элемент диапазона.

(2...100).begin #=> 2

**#end** — этот метод возвращает последний элемент диапазона, пример:

(1..100).end #=> 100

(1...100).end #=> 100

('a'..'z').end #=> "z"

('a'...'z').end #=> "z"

**#exclude\_end** — данный метод возвращает true для диапазонов, которые не включают максимум (диапазоны с тремя точками) и false для диапазонов, которые включают максимум, пример:

(1..10).exclude\_end? #=> false

(1...10).exclude\_end? #=> true

***Итерация по диапазону***

**#each** — сей метод позволяет вам проходить по каждому элементу входящему в диапазон и выполнять с ним определенные действия, например:

|  |
| --- |
| (1..10).each{|e| print e, ', '} #1, 2, 3, 4, 5, 6, 7, 8, 9, 10,  => 1..10 |

**#step** — данный метод нам уже знаком, так как мы уже сталкивались с ним, когда разбирались с целыми числами. Его отличие от целочисленного #step состоит в том, что обе границы заданы и нам необходимо задать лишь шаг итерации, пример:

('a'..'z').step(2){|ch| print ch, ", "} # a, c, e, g, i, k, m, o, q, s, u, w, y,  => "a".."z"

На этом с методами покончено. Важно понимать, что диапазон не является коллекцией, он ничего в себе не хранит, а просто указывает диапазон допустимых значений. Диапазон является самостоятельным типом данных, который покрывается классом **Range**:

(1..10).class #=> Range

Диапазоны очень полезны, например, они могут здорово помочь тогда, когда нам необходимо создать массив последовательных элементов:

(1..10).to\_a #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Представьте, что элементов не 10, а тысяча и вы поймете, что диапазоны действительно удобны. Разумеется, такой массив можно было бы создать при помощи цикла, однако это лишний, ухудшающий читабельность код!

Диапазоны очень удобны в использовании, с выражением **case**, пример:

a = 101

case a

when 1...50 then puts "0 < a < 50"

when 50...100 then puts "50 <= a < 100"

when 100...150 then puts "100 <= a < 150"

else

puts "a > 150"

end

# => 100 <= a < 150

Если бы не диапазон, вы бы не смогли написать такой простой и красивый код.

### ****Коллекции в Ruby: массивы и ассоциативные массивы (хэши)****

Коллекцией называется любой набор элементов. Коллекция представляет собой объект определенного типа (экземпляр определенного класса), который состоит из других объектов. Если коллекция содержит в себе другие коллекции, то такая коллекция называется ветвещейся колекцией или деревом или Root — коллекцией (корневой коллекцией), так как из нее происходит ветвление. Двумя основными коллекциями в Ruby являются массивы и хэши.

### ****Массив****

Массив представляет собой банковский сейф, где каждый элемент хранится в собственном бронированом ящике доступ к которому можно получить только имея ключь. Таким образом, данные представленные в массиве имеют парную структуру ключь — значение, в которой ключь служит для идентификации элемента в массиве и может быть только целым числом. Стоит заметить, что ключей (индексов) у каждого элемента массива имеется целых два: левосторонний и правосторонний, как и у строк. Примеры массивов:

[] #=> []

array = Array.new #=> []

array = %w{} #=> []

array = Array[]

Только что мы создали четыре пустых массива. Как видно из второй строки вышеприведенного примера массив в Ruby является объектом типа **Array**,то есть экземпляром класса **Array**. В примерах выше показано четыре основных способа создания массива, сразу скажу, что способ с Array.new применяется редко так как имеет избыточный синтаксис, хотя очень часто его использование имеет явную выгоду, вместо него следует стараться использовать [] — это самый короткий способ создания нового массива. Давайте создадим массив наполненные элементами:

array = ["Welcome", "love", "to", "Ruby!"]

Только что мы создали массив, который содержит четыре элемента строкового типа. Ruby язык высокого уровня, в котором в отличие от, например, Си не нужно заранее определять размерность массива и тип хранимых в нем данных. Массивы в Ruby могут хранить произвольное количество элементов, которое ограничено лишь оперативной памятью вашего компьютера, при этом данные содержащиеся в массиве могут иметь совершенно разные типы! Давайте добавим в наш массив несколько новых значений тех типов данных, с которыми мы уже знакомы:

array[4] = "STRING"

array[5] = (2..5)

array[6] = 12

array[7] = 5.78

array[8] = [1, 2, 3, "hello"]

Только что, мы добавляли значения в массив по предварительно определенным для ним ключам. Под ключем 0 у нас содержится строка «Welcome», а под ключем 8 — вложенный массив [1, 2, 3, "hello"]. Когда после имени массива мы пишем квадратные скобки в которых указываем число N, а затем ставим знак «=» — это называется добавление массиву нового элемента, который будет иметь ключь (индекс) — число 4. Чтобы обратиться к элементу массива следует использовать тот же синтаксис, но без знака «=», пример:

array #=> ["Welcome", "love", "to", "Ruby!", "STRING", 2..5, 12, 5.78, [1, 2, 3, "hello"]]

array[0] #=> "Welcome"

array[5] #=> 2..5

array[8] #=> [1, 2, 3, "hello"]

Чтобы помещать в массив новые элементы, не обязательно заранее определять ключь, он будет определен автоматически, если вы будете использовать метод <<, пример:

array #=> ["Welcome", "love", "to", "Ruby!", "STRING", 2..5, 12, 5.78, [1, 2, 3, "hello"]]

array << "new element" #=> ["Welcome", "love", "to", "Ruby!", "STRING", 2..5, 12, 5.78, [1, 2, 3, "hello"], "new element"]

Как видно из примера, новый элемент добавляется в конец массива и ему присваивается индекс на 1 больше, чем у предыдущего элемента.

Вы можете помещать значения в массив не по порядку, пример:

|  |
| --- |
| a = []  a[0] = 1  a[5] = 2 |

При этом будет создан массив не с двумя ключами, а с шестью, при этом промежуточные значения между 0 и 5 будут иметь значение nil, пример:

a #=> [1, nil, nil, nil, nil, 2]

Таким образом у нас нет какой-либо последовательности, имеется значение с ключем (индексом) 1 и значение с ключем 1000. Единственная закономерность состояит в том, что элементы массивов, сортируются по ключам, то есть чем меньше ключь, тем значение ближе к началу массива, а чем ключь больше, тем ближе к концу.

Прежде, чем мы окунемся в основные методы для работы с массивами, хочу разобрать с вам такой синтаксис объявления массивов, как идентификатор массива %w, который очень похож на то, с чем уже познакомились изучая строки — идентификатором строки %q и другими способами создания массива. Идентификатор массива %w так же позволяет использовать любые пары символов, кроме букв и чисел для помещения в них содержимого массива. Использование синтаксиса объявления массива с идентификатором %w подходит в тех случаях, когда вы пытаетесь поместить в массив некоторую последовательность элементов разделенных одним лишь пробелом, при этом каждый элемент массива созданного в такой способ будет иметь строковый тип, пример:

a = %w{Welcome love to Ruby! 1 2 [1,2,3]}

#=> ["Welcome", "love", "to", "Ruby!", "1", "2", "[1,2,3]"]

Однако, элементы добавляемые после объявления массива, не будут конвертироваться в строку:

|  |
| --- |
| a << [10, 100]  #=> ["Welcome", "love", "to", "Ruby!", "1", "2", "[1,2,3]", [10, 100]] |

Выше я упоминал о том, что создания массива при помощи Array.new следует избегать, однако есть случаи, когда такой синтаксис может быть очень полезен. Метод **.new** класса **Array** способен принимать блок кода, который возвращает элементы помещаемые в массив, таким образом мы можем сэкономить несколько строк кода за счет расчета элементов массива в одной строке с его объявлением, пример:

a = Array.new(10){|elem| elem.odd? ? elem\*\*2 : elem\*\*3}

#=> [0, 1, 8, 9, 64, 25, 216, 49, 512, 81]

В такой простой способ мы создаем массив в десятью элементами, при этом каждый четный элемент содержит значение K в кубе, а каждый нечетный — K в кубе, где K — индекс элемента. Генерация элементов в блоке — это единственный повод использовать синтаксис Array.new, однако многие ссылаются на то, что такой синтаксис более явно указывает на то, что объект, на который ссылается переменная является массивом. Такое утверждение меня, честно говоря удивляет, поскольку квадратные скобки также не двусмысленно намекают на то, что объект — массив. Синтаксис array = Array[] используется также в целях явного указания типа объекта, что лично я считаю необоснованным.

**Работа с массивами**

Создание массива:

array = [1,2,3]

Обращение к элементам массива

array[0] #=> 1

array[2] #=> 3

array[3] #=> nil

Кроме непосредственно индексов вы можете использовать диапазоны ключей или их перечень, примеры:

array[0,2] #=> [1, 2]

array[0..2] #=> [1, 2, 3]

Ruby очень мощный язык программирования еще и потому, что позволяет вставлять во многие выражения другие выражения, такая вложенность позволяет писать более короткий код, при этом в руках и глазах более-менее опытного программиста код не теряет читабельности, пример:

array[1..array.size] #=> [2, 3]

**#first**, **#last** — данные методы служат для получения первого и последнего элементов массива и унаследованы от модуля Enumerable, пример:

array.first #=> 1

array.last #=> 3

**#pop** — данный метод возвращает последний элемент массива (элемент с наибольшим ключем) и удаляет его из массива, пример:

array.pop #=> "ruby"

array #=> [1, 2, 3, "hello!", "...and bye!"]

**#take** — сей метод получает в качестве аргумента число N и возвращет N — первых элементов массива, пример:

[1,5,6,7,10].take(3) #=> [1, 5, 6]

**#take\_while** — данный метод выбирает из массива по одному элементу и передает его в блок кода, пока блок кода не возвращает значение false, метод возвращает элементы массива, пример:

[1,2,3,4,5,4,3,2,1].take\_while{|elem| elem < 5} #=> [1, 2, 3, 4]

В примере выше мы возвращали все элементы массива до тех пор, пока не наткнулись на первый элемент, который не соответствует условию в блоке кода.

**#values\_at** — очень мощный метод, который позволяет получить массив элементов из массива для которого метод был вызван указав набор индексов, диапазон индексов или набор диапазонов индексов элементов, пример:

array = [1,2,3,"hello", [10,20,30]]

array.values\_at(1,2,3) #=> [2, 3, "hello"]

array.values\_at(1,2,-3) #=> [2, 3, 3]

array.values\_at(1,2,-2) #=> [2, 3, "hello"]

array.values\_at(0..2,-1) #=> [1, 2, 3, [10, 20, 30]]

array.values\_at(0...2,2..4) #=> [1, 2, 3, "hello", [10, 20, 30]]

Добавление элементов в массив:

array[3] = "hello!"

array[3] #=> "hello!"

array << '...and bye!'

array #=> [1, 2, 3, "hello!", "...and bye!"]

**#push** — данный метод выполняет абсолютно ту же работу, что и метод **<<**, то есть добавляет элемент в конец массива, пример:

array.push("ruby")

#=> [1, 2, 3, "hello!", "...and bye!", "ruby"]

**#insert** - очень полезный метод, который принимает пары аргументо ключь — значение и вставляет значения в массив с указанным ключем, при этом происходит сдвиг всех существующих элементов массива, пример:

array = [1,2,3,4,5]

array.insert(0,100) #=> [100, 1, 2, 3, 4, 5]

array.insert(0, 1, 1, 13) #=> [1, 1, 13, 100, 1, 2, 3, 4, 5]

**#unshift** — данный метод получает набор значений и помещает их в начало массива, при этом имеющиеся в массиве элементы сдвигаются в право, то есть их левосторонние индексы увеличиваются, пример:

a = [1,2,3,4,5]

a.unshif("a", "b", "c")

a.unshift("a", "b", "c") #=> ["a", "b", "c", 1, 2, 3, 4, 5]

a #=> ["a", "b", "c", 1, 2, 3, 4, 5]

***Подсчет элементов массива***

**#size** и **#length** — данные методы служат для возвращения количества элементов массива:

[1,2,3,4].length #=> 4

[1,2,3,4].size #=> 4

array = []

array [1] = 1

array.size #=> 2

array.length #=> 2

array[1000] = 'one thouthand'

array.size #=> 1001

array.length #=> 1001

**#count** — данный метод принадлежит  модулю **Enumerable**, о котором мы поговорим позже в этой главе. Метод #count используется также, как #size и #length, однако может выполнять параметризированный подсчет, пример:

array = [1,2,3,4,5,6,7,1,1,2,3]

array.count #=> 11

array.count(1) #=> 3

array.count(2) #=> 2

array.count{|elem| elem.even?} #=> 4

array.count{|elem| elem >= 5 } #=> 3

Мы можем использовать #count для подсчета реально определенных элементов массива, то есть тех, что не содержат значение nil и для других полезных целей:

array = []

array[1] = 1

array[1000] = 2

array.count #=> 1001

array.count{|elem| elem.nil?} #=> 999

#метод #nil? возвращает true, если значение nil

array.count{|elem| !elem.nil?} #=> 2

***Сортировка массива***

**#sort** - данный метод служит для сортировки массива, если быть совсем точным, то данный методвозвращает новый массив с отсортированными элементами, а для сортировки самогого массива следует использовать соответствующий BANG метод **#sort!**, примеры:

array = [4, 5, 7, 8, 1, 5, 3]

array.sort #=> [1, 3, 4, 5, 5, 7, 8]

array #=> [4, 5, 7, 8, 1, 5, 3]

array.sort! #=> [1, 3, 4, 5, 5, 7, 8]

array #=> [1, 3, 4, 5, 5, 7, 8]

["a", "b", "e", "d", "c"].sort #=> ["a", "b", "c", "d", "e"]

["a", "b", "e", "d", "c", 1, 4, 2, 3].sort #=> ArgumentError: comparison of String with 3 failed

Сортировка успешно справляется как с чисами, так и со строками, однако при комбинации типов возникают неприятности, которые однако можно легко решить самописным методом, который будет предоставлен в конце данной главы.

Мало кто знает, что метод #sort позволяет также сортировать вложенные массивы, давайте рассмотрим пример такого использования:

array = []

array << [1,5] << [1,3,4] << [2,5,6] << [1,2] << [12,6,8] << [1,7] << [1,2,3] << [1,1,2]

array.sort! #=> [[1, 1, 2], [1, 2], [1, 2, 3], [1, 3, 4], [1, 5], [1, 7], [2, 5, 6], [12, 6, 8]]

Как видно из примера сортировка элементов — массивов происходит сразу по двум критериям: по N — элементу массива и количеству элементов. Массив с наименьшим значением N элемента будет всегда ближе к началу root-массива (родительского массива), чем вложенный массив с большим значением соответствующего по порядку элементов, если во вложенных масивах порядок следования элементов равен, то первым будет тот, который содержит меньшее число элементов. Такой порядок сортировки свойственен, например, различным словарям.

Метод #sort может выполнять как сортировку от меньшего к большему, так и наоборот, для этого методу #sort необходимо передать блок кода, в которому будет описано правило, например:

array = [1,6,7,9,2,3,7,4]

array.sort{|a,b|a<=>b} #=> [1, 2, 3, 4, 6, 7, 7, 9]

array.sort{|a,b|b<=>a} #=> [9, 7, 7, 6, 4, 3, 2, 1]

Метод #sort получает из массива два рядом идущих элемента a и b и сравнивает их при помощи уже знакомого вам метода <=>, из результатом сравнения становится понятно, следует ли поменять позиции элементов, или необходимо оставить каждый на своем месте.

**#sort\_by** — очень мощный метод, который предоставляет сортировку массива по определенному параметру, который передается в блоке, примеры:

[1,-6,2,-4,3].sort\_by{|a| a} #=> [-6, -4, 1, 2, 3]

[1,-6,2,-4,3].sort\_by{|a| a\*\*2} #=> [1, 2, 3, -4, -6]

Метод #sort\_by получает по одному элементу массива и выполняет и выполняет с каждым из них определенное действие, по результату которого, происходит сортировка, еще пример с сортировкой по количеству символов в строке:

['New York','Atlanta', 'Washington', 'Detroit'].sort\_by{|string| string.length}

#=> ["Detroit", "Atlanta", "New York", "Washington"]

**#reverse** — данный метод возвращает новый массив с элементами отсортированными в противоположном порядке, пример:

a =[1,2,3] #=> [1, 2, 3]

a.reverse #=> [3, 2, 1]

a #=> [1, 2, 3]

a.reverse! #=> [3, 2, 1]

a #=> [3, 2, 1]

**#shuffle** — выполняет перемешивание элементов массива случайным образом, пример:

a = [1,2,3,4,5]

a.shuffle #=> [3, 4, 1, 2, 5]

a.shuffle #=> [3, 4, 5, 2, 1]

a #=> [1, 2, 3, 4, 5]

a.shuffle! #=> [2, 4, 5, 3, 1]

a #=> [2, 4, 5, 3, 1]

***Арифметика множеств***

**+** — данный метод позволяет произвести сложение двух массивов в один, элементы прибавляемого массива будут добавлены в конец новообразованого пример:

[1,2,4,5, "hello"] + [5,6,[1,2,3]] #=> [1, 2, 4, 5, "hello", 5, 6, [1, 2, 3]]

**\*** — умножение массива, пример:

[1,2,3]\*3 #=> [1, 2, 3, 1, 2, 3, 1, 2, 3]

**-** — разность массивов A — B, в результате этой операции возвращается массив уникальных элементов массива A, пример:

[1,2,3,3,4,4,5] - [1,2,3,5,6] #=> [4, 4]

**<=>** — сравнение массивов как множеств, сравнивая массивы A и B мы можем получить три результата сравния: 1 — множество A включает множества B, 0 — множество A и B равны, -1 — множество B включает один и более элементов не входящих в множество A. Стоит упомунять, что порядок следования элементов имеет значение, то есть [1,2] и [2,1] не будут считаться одинаковыми, примеры:

[1,2,3,5] <=> [1,2,6] #=> -1

[1,2,3,5] <=> [1,2] #=> 1

[1,2,3,5] <=> [1,2,3] #=> 1

[1,2,3,5] <=> [1,2,3,5] #=> 0

[1,2,3,5] <=> [1,2,5,3] #=> -1

**==** — строгое сравнение массивов, строгим я его называю потому, что такое сравнение вернет true только тогда, когда массивы имеют одинаковые элементы, одинаковое их количество и одинаковый порядок следования, во всех остальных случаях будет получено значение false, пример:

[1,2,3] == [1,2,3] #=> true

[1,2,3] == [1,2,3,4] #=> false

[1,2,3] == [1,2,4] #=> false

[1,2,3] == [1,2] #=> false

**&** — пересечение множеств, данный метод возвращает массив совпадающих элементов в нескольких множествах, примеры:

[1,2,6,9,100] & [2,34,100] #=> [2, 100]

[1,2,6,9,100] & [2,34,100] & [1,100] #=> [100]

***Удаление и чистка массива***

**#shift** — метод возвращает значение первого элемента массива и удаляет его, пример:

a = [1,2,3,4]

a.shift #=> 1

a #=> [2, 3, 4]

**#delete\_at** - метод принимает индекси элементов, которые следует удалить и удаляет их, пример:

a = [1,2,3,4,5]

a.delete\_at(1) => 2

a.delete\_at(0) #=> 1

a.delete\_at(2) #=> 5

a #=> [3, 4]

**#drop** — метод используется для возвращения нового массива с удаленным N — числом элементов с начала массива, пример:

a = [1,2,3,4,5,"bingo"]

a.drop(3) #=> [4, 5, "bingo"]

a.drop(a.size-1) #=> ["bingo"]

a #=> [1, 2, 3, 4, 5, "bingo"]

**#drop\_while** — данный метод похож на метод #drop c тем лишь отличием, что вместо удаления N-количества первых элементов, он получает блок кода с условием и удаляет элементы начиная с начала массива до тех пор, пока условие в блоке верно, пример:

a = [1,2,3,4,5]

a.drop\_while{|elem| (elem\*\*2) < 25} #=> [5]

a.drop\_while{|elem| (elem\*\*2) < 9} #=> [3, 4, 5]

**#delete** — данный метод принимает пример объекта для удаления и удаляет все соответствующие примеру элементы массива, пример:

a = ["hello",1,2,3]

a.delete("hello") #=> "hello"

a #=> [1, 2, 3]

**#delete\_if** - удаление элементов массива. Данный метод пробегается по массиву передавая его элементы в блок кода, когда блок кода возвращает true, то элемент удаляется, пример:

cities = ["California", "Washington", "Detroit", "New York", "Philadelphia"]

cities.delete\_if{|city| city.length > 8} #=> ["Detroit", "New York"]

**#uniq** — данный метод производит удаление всех не уникальных элементов массива, таким образом каждое значение в массиве содержится только один единственный раз, пример:

a = [1,2,3,3,2,5]

a.uniq #=> [1, 2, 3, 5]

a #=> [1, 2, 3, 3, 2, 5]

a.uniq! #=> [1, 2, 3, 5]

a #=> [1, 2, 3, 5]

**#compact** — данный метод позволяет удалить все элементы массив значение которых nil, пример:

a = []

a[0] = 1 #=> 1

a[5] = 2 #=> 2

a #=> [1, nil, nil, nil, nil, 2]

a.compact #=> [1, 2]

a #=> [1, nil, nil, nil, nil, 2]

a.compact! #=> [1, 2]

a #=> [1, 2]

**#clear** — данный метод произвоодит полную зачистку массива, пример:

a = [1,2,3,4,5]

a.clear

a #=> []

***Особое форматирование массивов***

**#flatten** — метод производит «схлопывание» массива. Под «схлопыванием» массива подразумевается приведение многомерного массива в одномерный вид, пример:

a= [1,2,[3,4,[5,6,7],[8,9]],[10,[11,12]]]

a.flatten #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

a.flatten(1) #=> [1, 2, 3, 4, [5, 6, 7], [8, 9], 10, [11, 12]]

a.flatten(2) #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

a #=> [1, 2, [3, 4, [5, 6, 7], [8, 9]], [10, [11, 12]]]

a.flatten! #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

a #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

В качестве аргумента метод #flatten принимает максимальный уровень вложенности массива, который необходимо «схлопнуть».

**#transpose** — транспонирование массива. Еси представить двуметный массив в виде таблицы, где вложенные массивы являются столбцами, то метод #transpose предоставляет возможность изменьсть структуру массива, на такую, где столбцы преобразуются в стоки. Словами объяснить это сложновато, вам следует взглянуть на пример, чтобы понять лучше:

a = [[10,11,12],[20,21,22]]

a.transpose #=> [[10, 20], [11, 21], [12, 22]]

Вместо двух столбцов по три строки образовалось три строки по два столбца.

### ****Ассоциативный массив — хэш****

Ассоциативный массив, или хэш — это очень удобная структура данных, которая предоставляет более удобный интерфейс обращения к данным. Все хэши имеют тип **Hash**, то есть являются экземплярами класса **Hash**. ниже прриведены способы объявления хэша:

a = {} #=> {}

a = Hash.new #=> {}

a = Hash[] #=> {}

Основными отличиями хэша от массива является то, что в хэше ключи должны быть присвоены элементу явно, кроме того, ключем может быть любой объект, а не только целое число, как то есть в массивах. Пример хэша:

hash = {'hello' => 'goodbye', 1 => "Hey!", [1]=>[1,2,3,4,5]} #=> {"hello"=>"goodbye", 1=>"Hey!", [1]=>[1, 2, 3, 4, 5]}

hash['hello'] #=> "goodbye"

hash[1] #=> "Hey!"

hash[[1]] #=> [1, 2, 3, 4, 5]

hash[[1]][1] #=> 2

Как видите, хэши позволяют также хранить любые типы данных и могут быть произвольного размера, который ограничивается только оперативной памятью компьютера, кроме того, ключи также могут иметь различный тип. Работа с хэшами на очень похожа на работу с массивами, поэтому мы не будем рассматривать их так же подробно.

Вам следует помнить, что хотя массивы и позволяют хранить элементы различных типов, это не является самым хорошим стилем. Вам следует использовать массивы для хранения более-менее одинаковых элементов или вложенных структур, которые в свою очередь так же могут быть массивами или хэшами. В идеале ваши структуры должны иметь следущий вид:

fruits = []

fruits << {"name"=>"banana", "cost"=>10} << {"name"=>"apple", "cost"=>7}

#=> [{"name"=>"banana", "cost"=>10}, {"name"=>"apple", "cost"=>7}]

Только что мы получили массив однородных сущностей — фруктов, именно для хранения однородных сущностей наиболее всего подходят массивы, поскольку не обладают «говорящими» индексами (ключами) элементов. Хэши — это те же массивы, но обладающие «говорящими» ключами, то есть такими, которые описывают информацию, которая за ними скрывается в отличие от индексов в массивах, которые просто указывают порядок следования элементов.

Прежде, чем мы перейдем к использованию хэшей, мы основим еще один полезный тип данных в Ruby — символы.

### ****Символ****

Символы — это специфичные строки, которые, тем не менее имеют поведение в чем-то похочее на числа. Ниже приведен простой пример символа:

:symbol.class #=> Symbol

Как видите, символ — это просто строка, без кавычек, которая начинается с символа двоеточия и которая имеет тип **Symbol**. Причина, почему были введены символы, когда есть строки заключается в том, что хранение символов в памяти компьютера происходит так же, как и хранение чисел, что позволяет экономить потребление операционной памяти и выремя выполнения программы, примеры:

'hello world'.object\_id #=> 70115500

'hello world'.object\_id #=> 70106870

'hello world'.object\_id #=> 70104250

'hello world'.object\_id #=> 70101670

:hello\_world.object\_id #=> 186168

:hello\_world.object\_id #=> 186168

:hello\_world.object\_id #=> 186168

5.object\_id #=> 11

5.object\_id #=> 11

5.object\_id #=> 11

Метод **#object\_id** возвращает уникальный идентификатор объекта — настоящее имя объекта, по которому к нему можно обратиться в оперативной памяти. Как видите, у одинаковых строк ID разный, а значит и каждая строка, хоть и имеет одинаковое содержимое представленна своим собственным объектом, поэтому, для хранения одной и той же информации в строке сосдается целых четыре объекта. У чисел и символов все иначе, одинаковые числа и символы не плодят новых объектов и это видно из того, что ID объекта для одинаковых чисел и символов одинаков. Такое поведение символов делает выгодным их использование в качестве ключей для хэшей. Почему? — Вы только представьте хэш состоящий из нескольки миллионов пар ключь-значение, где ключем является строка. В результате создания такого хэша нам понадобится, во первых, создать несколько миллионов объектов только для хранения ключей, да еще и хранить их в оперативной памяти, как результат, оперативной памяти может не хватить, а генерация огромного количество объектов может занять большое количество времени, ровно, как и уничтожение объектов после того, как необходимость в их использовании отпадает. В таком случае нам следует использовать символы и вместо миллионов объектов мы можем хранить 1 — 10 или N объектов в зависимости от структуры хэша, что несоизмеримо меньше с миллионами объектов при использовании строк.

Бытует мнение о том, что символы использовать вредно, так как они не удаляются сборщиком мусора в Ruby. (Сборщик Муссора — специальный компонент в языках программирования, который занимается тем, что «убивает» объекты хранящиеся в оперативной памяти, которые более в ходе выполнения программы не понядобятся). Сразу хочу сказать, что это действительно так и объекты ключей хранятся до самого конца выполнения программы, однако проблема надумана, поскольку хранение нескольких десятков или сотен объектов символьного типа, хоть и более продолжительное, однако и более экономное, чем хранение тысячь или миллионов объектов типа String. Кроме того, использование символов более удобное, поскольку не нужно использовать надоедливые кавычки, да и в Ruby 1.9 появился новый синтаксис для ключей символьного типа, который делает код еще более читабельным, вот он:

#старый синтаксис:

hash = {:cost => 150, :weight => 200, :color => "green"}

#новый, введенный в Ruby 1.9 (доступен только для символьных ключей):

hash = {cost: 150, weight: 200, height: "green"}

Как видите, новый синтаксис более удобен, поэтому я рекомендую использовать именно его, хотя моей рекомендации найдется много людей, которые будут возражать аргументируя это тем, что такое код не будет совместим со старыми версиями Ruby, где после ключа обязательно должен был следовать «хэш-рокет (=>)».

Символ -> Строка -> Символ

**#to\_s** - данный метод используется для преобразования объекта в строку, если быть совсем точным, то объект преобразовываться в другой тип не может, просто создается другой, аналогичный объект — строкового типа, пример:

:symbol.to\_s #=> "symbol"

**#to\_sym** — сей метод предназначен для получения соответствующего строке символа, пример:

"string".to\_sym #=> :string

Еще один малоизвестный способ преобразовать строку в символ — просто поставить перед строкой двоеточие, данный способ, вполне может заменить метод to\_s в тех случаях, когда используется непосредственно для строки, а не для переменной ссылающейся на нее, примеры:

symbol = "string".to\_sym #=> :string

symbol.object\_id #=> 54088

symbol2 = :"string" #=> :string

symbol2.object\_id # => 54088

***Преобразование символа в процедуру***  
Ruby является не просто объектно-ориентированным языком программирования, но поддержиивает и другие парадигмы, например функциональную. Таким образом имеет в своем арсенале процедуры, которые поначалу лучше воспринимать как методы, которые являются объектами. У символов есть очень мощный и полезный метод #to\_proc, который позволяет создать из символа процедуру. Созданная в такой способ процедура является ни чем иным, как вызовом в блоке кода методо соответствующего символу, примеры:

cap = :capitalize.to\_proc #=> #<Proc:0x9624040>

cap.call("hello world") #=> "Hello world"

proc = :blablabla.to\_proc #=> #<Proc:0x962bcdc>

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)}, но это пока лишние подробности, с которыми вы ознакомитесь в отдельной главе посвященной методам и процедурам, а пока просто знайте, что **#call** используется для вызова процедуры (ее выполнения), а **#send** для вызова соответствующего символу метода у объекта — приемника, «hello».send(:upcase) равносильно «hello.upcase».

**#upcase**, **#downcase**, **#capitalize**, **#swapcase** — данные методы символов соответствуют аналогичным методам строк и возвращают новые объекты символьного типа с соответствующими изменениями, пример:

:symbol.upcase #=> :SYMBOL

:SYMBOL.downcase #=> :symbol

:symbol.capitalize #=> :Symbol

:symbol.swapcase #=> :SYMBOL

:SYMBOL.swapcase #=> :symbol

Важно знать: у символов отсутствуют BANG методы так как объект символьного типа не может изменяться в отличие от строки, так как строка является ни чем иным, как коллекцией букв, цифр и прочих символов.

Вот мы и разобрались с тем, что же это за тип такой — символы, и возвращаемся к изучению хэшей.

**Использование хэшей**

Хэши очень любят символы, об этом я уже упоминал выше, когда писал о том, зачем были придуманы эти самые символы. Поскольку, хэши имеют достаточно много общих методом в массивами, я не буду зацикливать внимание на  подробном их разборе, просто приведу примеры работы с хэшами, ну и остановлюсь, на некоторых важных и уникальных методах.

Внимание: в данном учебнике я использую синтаксис хэшей. который был введен в Ruby 1.9, поэттому в более старых версиях у вас будут возникать ошибки. Я надеюсь, вы послушались моего совета в первой главе данного учебника и установили Ruby 1.9.2. Если же вы используете, например, Ruby 1.8.7, то вам необходимо несколько изменять синтаксис создания хэшей, все это описано в этой главе в разделе посвященному символам.

user = {name:"Vasya", last\_name: "Petrov", age: 20}

user[:name] #=> "Vasya"

user[:last\_name] #=> "Petrov"

user["last\_name".to\_sym] #=> "Petrov"

Синтаксис создания хэша с Hash.new позволяет принимать блок кода, который выполняется при обращении к несуществующщему ключю,  создает его и соответствующее ему значение по правилам определенным в блоке кода, пример:

hash = Hash.new{|h,k| h[k] = k} #=> {}

hash[:a] #=> :a

hash #=> {:a=>:a}

Другими словами, в такой способ можно создать значение по умолчанию.

Синтаксис Hash[] позволяет создавать хэш из предоставляемого массива. Такой синтаксис создания хэша, как бы просит у переданного массива расчитаться на 1й — 2й, где 1й — ключь, 2й — значение, пример:

hash = Hash[1,2,3,4,5,6] #=> {1=>2, 3=>4, 5=>6}

hash = Hash[:a,1,:b,2,:array, [1,2,3,4]] #=> {:a=>1, :b=>2, :array=>[1, 2, 3, 4]}

hash = Hash[[[:a,:b,:c,:d],[1,2,3,4]].transpose] #=> {:a=>1, :b=>2, :c=>3, :d=>4}

**#default** - данный метод позволяет установить хэшу значение по умолчанию. В массивах значение по умолчанию всегда nil, а в хэшах его можно очень просто установить, пример:

h=Hash.new #=> {}

h[:a] #=> nil

h.default = 'ruby'

h[:a] #=> "ruby"

h[:b] #=> "ruby"

**#invert** — данный метод позволяет преобразовать структуру хэша так, что ключи и значения меняются местами, пример:

h = {a: 100, b: 200, c: 300}

h.invert #=> {100=>:a, 200=>:b, 300=>:c}

**Массив -> Хэш -> Массив**

Для преобразования массива в хэш достаточно воспользоваться специальным синтаксисом создания хэша: Hash[], который разбивает массив на пары ключь — значение, примеры:

array =[1, 2, 3, 4]

hash = Hash[\*array] #=> {1=>2, 3=>4}

Обратите внимание на символ **\*** — это оператор носит название «splat» или «asterisk» и используется для преобразования массива в набор аргументов, пример:

array #=> [1, 2, 3, 4]

[array] #=> [[1, 2, 3, 4]]

[\*array] #=> [1, 2, 3, 4]

Для преобразования хэша в массив имеется специальный метод to\_a, пример:

hash #=> {1=>2, 3=>4}

hash.to\_a #=> [[1, 2], [3, 4]]

Чтобы конвертировать хэш в массив с такой структурой, как тот, из которого создавался хэш, необходимо добавить в цепочку методов знакомый нам #flatten:

hash.to\_a.flatten #=> [1, 2, 3, 4]

Опять таки, запомните, что говоря о преобразовании хэша в массив, я не имею введу преобразование самого хэша, я имею введу создание нового объекта — массива, для создания которого хэш — приемник метода to\_a аграет роль поставщика значений.

**Его величество модуль Enumerable и методы — итераторы**  
Ruby предоставляет такие контейнеры кода, как модули, они же примеси (mixin). О модулях мы поговорим в отдельной главе, а сейчас я расскажу об одном очень популярном модуле **Enumerable**, который 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** — данный итератор пробегает по коллекции и передает каждый ее элемент в блок кода, где над ними происходит определеное действие, примеры:

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, пример:

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** — данные метод предоставляет доступ не только к значениям элементов, но и их индексам, пример:

a.each\_with\_index{|value, key| a[key] = value \*\* 2} #=> [1, 4, 9, 16, 25]

a #=> [1, 4, 9, 16, 25]

При использовании #each\_with\_index вместе с диапазонами, порядок следования аргументов блока (ключа и значения) меняется, пример:

(1..5).each\_with\_index{|key, value| print v, "-", k, " "}

0-1 1-2 2-3 3-4 4-5

При выполнении #each\_with\_index для хэша происходит его преобразование в массив, пример:

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 -> [:d, 400]

**#collect** и **#map** — методы синонимы, которые используются для создания новой коллекции из уже имеющейся, примеры:

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, пример:

|  |
| --- |
| 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** — данный метод мы уже рассматривали в разделе посвященном массивам, однако давайте послушаемся старой пословицы «Повторение — мать учения»:

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 возвращаюст 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, примеры:

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 программируя на Rails метод #find у вас будет переписан одноименным методом из **ActiveRecord**. Методы #find и #detect используются для получения первого эллемента коллекции который соответствует условию в блоке кода, примеры:

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 жеработу, 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 уже упоминались в данной главе, они, как не странно, принадлежат строковым объектам и передают при каждой итерации по одному символу в блок кода, пример:

str = ""

"hello readers!".chars{|char| str << char.upcase}

str #=> "HELLO READERS!"

### ****Булевы типы данных(false и true) и значение nil****

Булевы типы данных **True** и **False** также являются объектами, экземплярами классов **TrueClass** и **FalseClass**, в Ruby к ним также относят значение **nil**, которое является экземпляром класса **NilClass** и интерпретируется как значение false, это сделано из расчета на то, чтобы мочь показать в коде не просто «ЛОЖЬ», но и «ПУСТОТУ» и при необходимости, при возвращении этих двух значений, производить различные действия.

**#nil?** — данный метод возвращает значение true, если объект имеет тип NilClass, пример:

array = []

array << 1

array[5] = 6

array #=> [1, nil, nil, nil, nil, 6]

array.select{|v| v.nil?} #=> [nil, nil, nil, nil]

Значения True и False являються в первую очередь результатом операций сравнения. Кроме того, в Ruby все возвращает значение и любое значение кроме false и nil интерпретируются как true. Если, например, какой-то метод возвращает true, или значение, которое интерпретируется как true, то это свидетельствует о том, что метод выполнил свою работу успешно, если же возвращено значение false — это свидетельствует о препятствии выполнению.

2 > 5 #=> false

2 < 5 #=> true

puts 'hello'#=> напечатает hello, и возвратит значение nil

В данной главе были рассмотрены далеко не все методы стандартных типов и далеко не все методы модуля Enumerable и далеко не все типы данных в Ruby. Много информации было упущено не только потому, что данный учебник не ставит целью превратится в справочник методов.

Как и обещал, привожу пример метода, который производит сортировку массивов с различными значениями:

def multisort(arr)

  array = []

  prom = {}

  result = []

  arr.each do |value|

    case value.class.to\_s

      when 'String' then array << "\"#{value}\""

      when 'Symbol' then array << ":#{value}"

      else

        array << value.to\_s

    end

  end

  array.each do |value|

    type = eval(value).class.to\_s.downcase.to\_sym

    case type

      when :fixnum, :bignum, :float then type = :numeric

    end

    if prom[type]

      prom[type] << value

    else

      prom[type] = [value]

    end

  end

  prom.sort.each{|k,value| value.each{|v| result << eval(v)}}

  return result

end

array = [1, 5, nil, true,"[stringgg", nil, false, true, 8.4, 2.3, 3, 6, "{aa", "bb", "ab", :aaaa, :baba, [1,2,3], [1,2,4], [2,3], [1], {1=>2, 3=>200, 5=>1},{1=>2, 2=>3}, {1=>2}]

print multisort(array) #[[1, 2, 3], [1, 2, 4], [2, 3], [1], false, {1=>2, 3=>200, 5=>1}, {1=>2, 2=>3}, {1=>2}, nil, nil, 1, 5, 8.4, 2.3, 3, 6, "[stringgg", "{aa", "bb", "ab", :aaaa, :baba, true, true]

Суть заключается в том, что многотипный массив разбивается на однотипные массивы, каждый из которых сортируется отдельно, а затем все склеивается вместе. Поскольку, некоторые типы, например хэши, не сортируются, пришлось преобразовывать все объекты в строки, которые сортируются отлично, а затем из строк обратно в свой собственный тип данных при помощи eval, что конечно же снижает скорость работы, однако позволяет писать достаточно короткий код. Разумеется, данный метод не является идеальным и не является панацеей, есть много способов его улучшить, но это не суть важно, поскольку он приведен здесь лишь с целью демонстрации.

