Ruby - Руководство пользователя

Оригинал: sophtanet.com

Ruby -- "простой объектно- ориентированный язык". Сначала это может показаться несколько странным, но он был спроектирован таким образом, чтобы программы на Ruby было легко и читать и писать. Это *Руководство пользователя* поможет вам научиться запускать и использовать Ruby, а также даст вам понимание природы Ruby, которое вы можете и не получить, прочитав Reference Manual.

Содержание

История документа

- Оригинальная версия на японском языке -- matz.
- Первоначальный перевод на английский -- GOTO Kentaro & Julian Fondren.

- Дальнейшие перевод и сопровождение -- Mark Slagell. Date stamp для данной версии -- 20010216.
- Akinori Musha, Laurent Julliard, Manpreet Singh и Robert Gustavsson помогли с отслеживанием ошибок и улучшением форматирования и представления. Спасибо!
- Перевод на русский -- Alexander Miatchkov (Александр Мячков) с разрешения Mark Slagell.

Back to top

Ruby -- руководство пользователя

Что такое Ruby?

Ruby -- это "интерпретируемый скриптовый язык для простого и быстрого объектно-ориентированного программирования". Что бы это значило?

интерпретируемый скриптовый язык:

- Возможность прямого осуществления системных вызовов
- мощная поддержка операций со строками и регулярными выражениями
- немедленная обратная связь во время разработки

простое и быстрое программирование:

- отсутствие необходимости объявления переменных
- переменные не типизированы
- простой и последовательный синтакс
- автоматическое управление памятью

объектно - ориентированное программирование:

- все является объектом
- классы, наследование, методы и т.д.
- singleton- методы
- mixin при помощи модулей
- итераторы и скобочные операции

а также:

- целые числа с различной разрядностью
- модель обработки исключительных ситуаций
- динамическая загрузка
- потоки

Если Вы не особо знакомы с представленными выше концепиями - не переживайте и читайте дальше. Мантра Ruby -- *быстро и просто*.

Back to top

Ruby -- руководство пользователя

Поехали!

Для начала вы захотите проверить, установлен ли у вас Ruby. После символа подсказки в командной строке (он обозначен здесь как "%", так что % набирать не надо), введите

```
% ruby -v
```

(-v дает указание интерепретатору вывести его версию), затем нажмите кнопку *Enter*. Если Ruby установлен, вы увидите нечто похожее на:

```
% ruby -v
ruby 1.6.1 (2000-09-27) [i586-linux]
```

Если Ruby не установлен, Вы можете попросить Вашего администратора установить его или сделать это сами , поскольку Ruby является свободным ПО без ограничений на инсталляцию и использование.

Теперь давайте поиграем с Ruby. Вы можете ввести код программы прямо в командной строке, используя параметр -e:

```
% ruby -e 'print "hello world\n"'
hello world
```

Более обычным является сохранение программы в файле.

```
% cat > test.rb
print "hello world\n"
^D
% cat test.rb
print "hello world\n"
% ruby test.rb
hello world
```

^D -- это комбинация *control-D*. вышеприведенное верно для UNIX. При использовании DOS, попробуйте так:

```
C:\ruby> copy con: test.rb
print "hello world\n"
^Z
C:\ruby> type test.rb
```

```
print "hello world\n"
C:\ruby> ruby test.rb
hello world
```

При написании более сложного кода, чем этот, Вы, скорее всего, захотите использовать настоящий текстовый редактор.

Некоторые на удивление сложные и полезные вещи могут быть сделаны с помощью миниатюрных программ, умещающихся в командной строке. Например, эта программа заменяет строку foo строкой bar во всех исходниках на C и в хидер-файлах в текущем рабочем каталоге, делая резервные копии первоначальных файлов в файлах с добавлением ".bak" в конце названия:

```
% ruby -i.bak -pe 'sub "foo", "bar"' *.[ch]
```

А эта программа работает как команда cat в UNIX (но работает медленнее, чем cat):

```
% ruby -pe 0 file
```

Back to top

Давайте напишем функцию для вычисления факториала. Математическое определение факториала от n следующее:

```
n! = 1                    (n==0)
   = n * (n-1)!           (иначе)
```

На Ruby это может быть записано как:

```
def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end
```

Вы можете заметить повторение в появлении end. Ruby из-за этого называют "Algol-like". (На самом деле, синтакс Ruby более повторяет синтакс языка под названием Eiffel.) Вы также можете заметить отсутствие оператора return. В нем нет необходимости,

поскольку функция в Ruby возвращает последнее вычисленное в ней значение. Использование оператора return допустимо, но необходимым не является.

Теперь давайте опробуем нашу функцию вычисления факториала. Добавление одной строчки кода дает нам рабочую программу:

```
Программа для нахождения факториала от числа
# запишите ее в файл fact.rb

def fact(n)
   if n == 0
      1
   else
      n * fact(n-1)
   end
end

print fact(ARGV[0].to_i), "\n"
```

ARGV здесь - это массив, содержащий аргументы из командной строки, а to_i преобразует строку символов в численное представление.

```
% ruby fact.rb 1
1
% ruby fact.rb 5
120
```

Будет ли программа работать для аргумента, равного 40? Это вызвало бы переполнение при вычислениях на калькуляторе...

```
% ruby fact.rb 40
815915283247897734345611269596115894272000000000
```

Работает. На самом деле, Ruby может справляться с любыми целыми числами, дозволенными размерами памяти вашей машины. Так что и 400! может быть посчитан:

```
% ruby fact.rb 400
64034522846623895262347970319503005850702583026002959458684
44594280239716918683143627847864746326467629435057503585681
08482981628835174352289619886468029979373416541508381624264
61942352307046244325015114448670890662773914918117331955996
44070954967134529047702032243491121079759328079510154537266
72516278778900093497637657103263503315339653498683868313393
52024373788157786791506311858702618270169819740062983025308
59129834616227230455833952075961150530223608681043329725519
48526744322324386699484224042325998055516106359423769613992
31917134063858996537970147827206606320217379472010321356624
61380907794230459736069956759583609615871512991382228657857
```

```
954936161765448045322200782581840084843641559122945427538480
035583745180226759000613995601455952061272111929181050324910
008000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
```

Мы не можем навскидку проверить правильность результата, но, должно быть, он правильный. :-)

Цикл ввод/обработка

Когда Вы запускаете Ruby без аргументов, он считывает команды со стандартного устройства ввода и исполняет их по окончании ввода:

```
% ruby
print "hello world\n"
print "good-bye world\n"
^D
hello world
good-bye world
```

Ruby также поставляется с программой под названием eval.rb, которая позволяет вводить код на Ruby с клавиатуре в интерактивном цикле, показывая по ходу дела результаты его обработки. Она будет широко использоваться до конца тьюториала.

Если у Вас ANSI-совместимый терминал (это наверняка так и есть, если вы работаете на каком-либо из клонов UNIX, в DOS вам потребуются установленные драйвера ANSI.SYS или ANSI.COM), вам стоит использовать расширенный eval.rb, который добавляет визуальную помощь в расстановке отступов, выдает предупреждения, обеспечивает подсветку синтаксиса. В ином случае ищите в каталоге sample в каталоге установкии дистрибутива версию без поддержки ANSI. Вот короткий пример сессии работы с eval.rb:

```
% ruby eval.rb
ruby> print "Hello, world.\n"
Hello, world.
   nil
ruby> exit
```

hello world выводится функцией print. Следующая строка, в данном случае nil, сообщает о результатах выполнения последней команды; Ruby не делает различий между *оператором* and *выражением*, так что обработка куска кода просто означает то же самое, что и его исполнение. В данном случае, nil показывает, что print не вернул никакого определенного значения. Заметьте, что из интерепретатора можно выйти при помощи команды exit, хотя ^D тоже работает.

Далее в этом руководстве "ruby>" означает приглашение для ввода нашей маленькой полезной программки eval.rb.

Back to top

Ruby работает как с числами, так и со строками. Строка может быть заключена в двойные ("...") или одинарные ('...') кавычки.

```
ruby> "abc"
   "abc"
ruby> 'abc'
   "abc"
```

Двойные и одинарные кавички в некоторых случаях производят разный эффект. Строка в двойных кавычках разрешает экранирование символов после лидирующего обратного слэша, а также вычисление вложенных выражений при использовании #{}. Строки в одинарных кавычках не производят такой интерпретации; что видим, то и получаем. Примеры:

```
ruby> print "a\nb\nc","\n"
a
b
c
   nil
ruby> print 'a\nb\nc',"\n"
a\nb\nc
   nil
ruby> "\n"
   "\n"
ruby> '\n'
   "\\n"
ruby> "\001"
   "\001"
ruby> '\001'
   "\\001"
ruby> "abcd #{5*3} efg"
   "abcd 15 efg"
ruby> var = " abc "
   " abc "
ruby> "1234#{var}5678"
   "1234 abc 5678"
```

Оперирование со строками в Ruby более умное и интуитивное, чем в C. Например, вы можете соединять строки при помощи +, или повторять строку несколько раз при помощи *:

```
ruby> "foo" + "bar"
   "foobar"
ruby> "foo" * 2
   "foofoo"
```

Объединение строк на C гораздо более неудобное, поскольку оно требует явного управления памятью:

```
char *s = malloc(strlen(s1)+strlen(s2)+1);
strcpy(s, s1);
strcat(s, s2);
/* ... */
free(s);
```

А вот при использовании Ruby Вам не приходится интересоваться тем, сколько занимает строка. Мы вообще освобождены от управления памятью.

Вот еще некоторые примеры того, что можно делать со строками.

Объединение:

```
ruby> word = "fo" + "o"
   "foo"
```

Повторение:

```
ruby> word = word * 2
   "foofoo"
```

Вычленение символов (Отметьте, что символы в Ruby являются целыми числами)

```
ruby> word[0]
   102            # 102 is ASCII code of `f'
ruby> word[-1]
   111            # 111 is ASCII code of `o'
```

(Отрицательный индекс начинает отсчет не с начала, а с конца строки)

Выделение подстрок:

```
ruby> herb = "parsley"
   "parsley"
```

```
ruby> herb[0,1]
   "p"
ruby> herb[-2,2]
   "ey"
ruby> herb[0..3]
   "pars"
ruby> herb[-5..-2]
   "rsle"
```

Проверка на равенство:

```
ruby> "foo" == "foo"
   true
ruby> "foo" == "bar"
   false
```

Замечание: в Ruby 1.0 приведенные выше результаты представлены в верхнем регистре, т.е. TRUE.

А теперь попробуем некоторые из этих свойств в работе. Это будет пазл "Угадай слово", хотя паззл, возможно, звучит чересчур величественно для нашей поделки ;-)

```
# сохраните в файл guess.rb
words = ['foobar', 'baz', 'quux']
secret = words[rand(3)]

print "guess? "
while guess = STDIN.gets
   guess.chop!
   if guess == secret
      print "You win!\n"
      break
   else
      print "Sorry, you lose.\n"
   end
   print "guess? "
end
print "The word was ", secret, ".\n"
```

Не обращайте пока что слишком много внимания на детали этой программы. Вот как выглядит вывод пазла:

```
% ruby guess.rb
guess? foobar
Sorry, you lose.
guess? quux
Sorry, you lose.
guess? ^D
```

```
The word was baz.
```

(С учетом вероятности успеха 1/3, могло бы быть и лучше)

Back to top

Давайте-ка скомпонуем более интересную программу. На этот раз мы будем проверять, удовлетворяет ли строка некоторому набору свойств, который мы будем называть *шаблоном*.

Вот некоторые из символов и их сочетаний, которые имеют специальное значение при использовании в шаблонах:

[]	задание диапазона (т.е., [a-z] означает букву в пределах от a до z)
\w	буква либо цифра; то же, что и [0-9A-Za-z]
\W	ни буква, ни цифра
\s	символ пробела; то же, что и [\t\n\r\f]
\S	не пробел
\d	символ цифры; то же, что и [0-9]
\D	не цифра
\b	забой (0x08) (только если в пределах заданного диапазона)
\b	граница слова (только если в пределах заданного диапазона)
\B	не граница слова
*	ноль или более повторений предыдущего
+	один или более повторений предыдущего
{m,n}	минимум m и максимум n повторений предыдущего
?	не более одного повторения предыдущего; то же самое, что и{0,1}
|	совпадение предыдущего либо последующего условия
()	группировка

Общепринятый термин для шаблонов, которые используют этот странный вокабуляр - *регулярные выражения*. В Ruby, как и в Perl, они обычно окружаются прямыми слэшами, а не двойными кавычками. Если Вы никогда раньше не работали с регулярными выражениями, они кажутся какими угодно, только не *регулярными*; тем не менее было бы мудро потратить некоторое количество времени на близкое знакомство с ними. Они дают краткую и выразительную мощь, которая освободит Вас от головной боли (и

написания многих строк кода) в тех случаях, когда необходимо произвести определение соответствия шаблону, поиск и другие манипуляции над текстовыми строками.

Пусть, например, мы хотим проверить, удовлетворяет ли строка следующему описанию: "Начинается с f в нижнем регистре, сразу за ней следует Буква в верхнем регистре, потом что угодно, при условии, что буквы в нижнем регистре отсутствуют." Если Вы опытный программист на C, то Вы уже, вероятно, написали в своей голове примерно дюжину строк кода, так?? Поверьте, это едва ли поможет Вам. Но в Ruby Вам нужен всего лишь один запрос на соответствие строки регулярному выражению /^f[A-Z][^a-z]*$/.

А как насчет "Содержит шестнадцатеричное число, заключенное в угловые скобки?" Пожалуйста:

```
ruby> def chab (s)   # "contains hex in angle brackets"
    |    (s =~ /<0[Xx][\dA-Fa-f]+>/) != nil
    | end
  nil
ruby> chab "Not this one."
  false
ruby> chab "Maybe this? {0x35}"    # wrong kind of brackets
  false
ruby> chab "Or this? <0x38z7e>"    # bogus hex digit
  false
ruby> chab "Okay, this: <0xfc0004>."
  true
```

Хотя регулярные выражения могут с первого взгляда вызывать недоумение, озадачивать, Вы быстро начнете получать удовлетворение от возможности выражаться столь экономно.

Вот маленькая программа, чтобы помочь Вам поэкспериментировать с регулярными выражениями. Запишите ее как regx.rb и запустите командой "ruby regx.rb" в командной строке.

```
# Необходим терминал с поддержкой ANSI !

st = "\033[7m"
en = "\033[m"

while TRUE
  print "str> "
  STDOUT.flush
  str = gets
  break if not str
  str.chop!
  print "pat> "
  STDOUT.flush
  re = gets
```

```
    break if not re
    re.chop!
    str.gsub! re, "#{st}\\&#{en}"
    print str, "\n"
  end
  print "\n"
```

Нужно ввести 2 строки - в первой должна содержаться строка, а во второй регулярное выражение, на соответствие которому и проверяется строка. После проверки строка выводится на экран, причем все ее совпавшие части печатаются в инверсном режиме. Не забивайте себе голову деталями сейчас, скоро будет произведен анализ этого кода.

```
str> foobar
pat> ^fo+
foobar
~~~
```

То, что выше написано красным, на экране выглядит как инвертированное. "~~~~~" для тех, кто пользуется браузером в текстовом режиме.

Давайте попробуем другие сочетания.

```
str> abc012dbcd555
pat> \d
abc012dbcd555
    ~~~     ~~~
```

Если то, что выводит программа, для Вас является неожиданностью, то посмотрите внимательно на таблицу вверху этой страницы: \d не имееет никакого отношения к символу d, вместо этого он означает одиночную цифру.

А что, если имеется несколько способов удовлетворить условиям, накладываемым шаблоном?

```
str> foozboozer
pat> f.*z
foozboozer
~~~~~~~~
```

foozbooz дается как совпадение вместо fooz, поскольку регулярные выражения выделяют максимально длинную строку, удовлетворяющую шаблону.

Вот например, шаблон, позволяющий вычленить поле "время", внутри которого для разделения используются двоеточия.

```
str> Wed Feb  7 08:58:04 JST 1996
pat> [0-9]+:[0-9]+(:[0-9]+)?
```

```
Wed Feb  7 08:58:04 JST 1996
             ~~~~~~~~
```

"=~" это оператор выполнения сравнения по шаблону; он возвращает позицию в строке, начиная с которой найдено совпадение, или nil в случае, если таковое не найдено.

```
ruby> "abcdef" =~ /d/
   3
ruby> "aaaaaa" =~ /d/
   nil
```

Back to top

Ruby -- руководство пользователя

Массивы

Вы можете создать *массив*, перечислив несколько его элементов в квадратных скобках ([]) и разделяя их запятой. Массивы в Ruby могут включать в себя элементы разных типов.

```
ruby> ary = [1, 2, "3"]
   [1, 2, "3"]
```

Массвы могут быть повторены или объединены точно так же, как и строки.

```
ruby> ary + ["foo", "bar"]
   [1, 2, "3", "foo", "bar"]
ruby> ary * 2
   [1, 2, "3", 1, 2, "3"]
```

Мы можем использовать индексы для того, чтобы сослаться на любую часть массива.

```
ruby> ary[0]
   1
ruby> ary[0,2]
   [1, 2]
ruby> ary[0..1]
   [1, 2]
ruby> ary[-2]
   2
ruby> ary[-2,2]
   [2, "3"]
ruby> ary[-2..-1]
```

```
   [2, "3"]
```

(Отрицательные индексы обозначают относительное смещение считая с не с начала массива, а с конца.)

Массивы могут быть преобразованы в строки и из строк, используя join и split соответственно:

```
ruby> str = ary.join(":")
   "1:2:3"
ruby> str.split(":")
   ["1", "2", "3"]
```

Хешы

Ассоциативный массив имеет элементы, доступ к которым осуществляется не по порядковому номену (индексу), а по ключу *keys*, который может быть значением любого типа. Такие массывы иногда называют *хеш* или *словарь*; в мире Ruby предпочитают название *хэш*. Хеш может быть создан путем перечисления пар в круглых скобках ({}). Для поиска чего-либо в хеше используется ключ, так как если бы вы использовали индекс для поиска в в массиве.

```
ruby> h = {1 => 2, "2" => "4"}
   {1=>2, "2"=>"4"}
ruby> h[1]
   2
ruby> h["2"]
   "4"
ruby> h[5]
   nil
ruby> h[5] = 10     # добавляем значение
   10
ruby> h
   {5=>10, 1=>2, "2"=>"4"}
ruby> h[1] = nil    # удаление значения
   nil
ruby> h[1]
   nil
ruby> h
   {5=>10, "2"=>"4"}
```

Back to top

Ruby -- руководство

Снова к простым примерам

пользователя

Давайте теперь рассмотрим код некоторых примеров, показанных выше. Для ссылок мы во всех скриптах будем использовать номера строк.

Факториал

Этот скрипт был представлен в главе "Простые примеры".

```
01  def fact(n)
02      if n == 0
03          1
04      else
05          n * fact(n-1)
06      end
07  end
08  print fact(ARGV[0].to_i), "\n"
```

Поскольку это первое рассмотрение, мы проанализируем отдельно каждую строку.

```
01  def fact(n)
```

В первой строке def - это оператор, используемый для определения функции (или, точнее, *метода*; о том, что такое метод, мы расскажем в последующих главах). Здесь он определяет, что функции fact передается один аргумент, обозначенный как n.

```
02      if n == 0
```

if предназначено для проверки условия. Если условие выполняется, то просчитывается следующая строка; иначе выполняется то, что следует за else.

```
03          1
```

Значение, возвращаемое if, равно 1, если условие выполняется.

```
04      else
```

Если условие не выполняется, то вычисляется блок кода отсюда до end.

```
05          n * fact(n-1)
```

Если условие ны выполняется, то результатом if будет значение, равное n умноженное на fact(n-1).

```
06      end
```

Первое "end" закрывает оператор if.

```
07  end
```

Второе "end" закрывает оператор def.

```
08  print fact(ARGV[0].to_i), "\n"
```

В этой строке происходит вызов функции fact() с передачей в нее значения, определенного в командной строке, затем выводится результат.

ARGV - это массив, содержащий аргументы, переданные в командной строке. Элементами ARGV являются строки, так что мы должны привести их к целым числам, используя to_i. Ruby, в отличие от Perl, не преобразует автоматически строки в целые.

Гммм... А что произойдет, если мы скормим программе в качестве параметра отрицательное число? Видите проблему? А сможете исправить?

Строки

Теперь мы рассмотрим программу-пазл из главы, посвященной строкам.

```
01  words = ['foobar', 'baz', 'quux']
02  secret = words[rand(3)]
03
04  print "guess? "
05  while guess = STDIN.gets
06      guess.chop!
07      if guess == secret
08          print "you win\n"
09          break
10      else
11          print "you lose.\n"
12      end
13      print "guess? "
14  end
15  print "the word is ", secret, ".\n"
```

В этой программе использована новая управляющая структура while. Выполнение кода между while и соответствующим end будт повторяться до тех пор, пока некоторое заданное условие будет оставаться верным.

rand(3) в строке 2 возвращает случайную величину в диапазоне от 0 до 2 включительно. Это случайное значение используется для определения одного из элементов массива words.

В строке 5 мы считываем одну строку из со стандартного ввода при помощи метода STDIN.gets. Если получаем *EOF* (end of file - конец файла) во время считывания, gets возвращает nil. Таким образом, выполнение блока внутри while продолжается до тех пор, пока не появится *^D* (или *^Z* в DOS), означающее конец ввода.

guess.chop! в строке 6 удаляет последний символ из guess; в нашем случае это всегда будет символ *newline* (новая строка).

В строке 15 мы выводим задуманное слово. Мы записали это в виде оператора print с тремя аргументами (которые выводятся друг за другом), но было бы так же эффективно записать его с одним аргументом, написав вывод secret в виде #{secret} с целью ясно обозначить то, что это переменная, вместо которой должно быть подставленоо ее значение, а не просто строка символов для вывода:

```
print "the word is #{secret}.\n"
```

Регулярные выражения

В конце концов рассмотрим программу из главы regular expressions.

```
01  st = "\033[7m"
02  en = "\033[m"
03
04  while TRUE
05      print "str> "
06      STDOUT.flush
07      str = gets
08      break if not str
09      str.chop!
10      print "pat> "
11      STDOUT.flush
12      re = gets
13      break if not re
14      re.chop!
15      str.gsub! re, "#{st}\\&#{en}"
16      print str, "\n"
17  end
18  print "\n"
```

В строке 4 вместо условия в while жестко зашито значение true, так что плучается нечто, выглядящее как бесконечный цикл. Тем не менее, мы вставляем оператор break в 8 и 13 строки для обеспечения возможности выхода из цикла. Эти два break также являются примерами модификаторов оператора if. "if-модификатор" позволяет выполнение кода слева от себя тогда и только тогда, когда заданное условие выполняется.

Добавление по поводу chop! (см. строки 9 и 14). В Ruby мы обычно добавляем '!' или '?' в конце названий определенных методов. Восклицательный знак (!, иногда поизносимый

как "bang!") означает нечто потенциально деструктивное, то есть нечто, способное изменить значение того, над чем оно работает. chop! прямо воздействует на исходную строку, но chop без восклицательного знака работает с копией. Ниже проиллюстрировано это различие.

```
ruby> s1 = "forth"
  "forth"
ruby> s1.chop!       # Изменяется s1.
  "fort"
ruby> s2 = s1.chop   # Измененная копия перекидывается s2,
  "for"
ruby> s1             # ... без воздействия на s1.
  "fort"
```

Позднее Вы столкнетесь с именами методов, оканчивающимися на вопросительный знак (?, иногда произносимый как "huh?"); это означает предикатный метод, который может возвратить true либо false.

Строке 15 необходимо уделить особое внимание. Во-первых, отметьте наличие еще одного так называемого деструктивного метода gsub!. Он изменяет в str все подходящие под шаблон re последовательности (sub означает замена - substitute, а лидирующее g означает глобальное, т.е., заменяются *все* подходящие части строки, а не только первую совпавшую). Чем дальше, тем лучше; но чем же мы заменяем совпавшие части текста? st и en были определены ANSI-последовательности для инвертирования текста и приведения его к обычному виду соответственно. В строке 15 они заключены в #{} для обеспечения того, что они будут интерпретированны именно таким образом (и мы не видим неправильно выведенные *имена* переменных). Между ними мы видим "\\&". Небольшая хитрость. Поскольку строка, в которой производится замена, прописана в двойных кавычках, то пара обратных слешей будет интерпретироваться как один обратный слеш; то, что gsub! получит на самом деле, будет выглядеть как "\&", и это оказывается особым кодом, который ссылается на первое, попавшее под шаблон. Таким образом, новая строка при печати будет выглядеть точно так же, как и старая, за исключением того, что участки, совпавшие с шаблоном, будут выведены в инверсном виде.

Back to top

Ruby -- руководство пользователя — Управляющие структуры

В этой главе более конкретно рассматриваются управляющие структуры Ruby.

case

Мы используем оператор case для проверки последовательности условий. Это примерно похоже на оператор switch в C и Java, но по сравнению с ним, как мы увидим далее, более мощный.

```
ruby> i=8
ruby> case i
    | when 1, 2..5
    |   print "1..5\n"
    | when 6..10
    |   print "6..10\n"
    | end
6..10
   nil
```

2..5 - это выражение, которое означает *диапазон* между 2 и 5 включительно. Следующее выражение означает проверку на вхождение значения i в диапазон:

```
(2..5) === i
```

case внутри использует оператор отношения === для проверки нескольких условий за раз. Придерживаясь объектно-ориентированной природы Ruby, === интерпретируется соответственно объекту, находящемуся в условии when. Например, следующий код проверяет строки на равенство в первом when, и соответствие регулярному выражению во втором when.

```
ruby> case 'abcdef'
    | when 'aaa', 'bbb'
    |   print "aaa or bbb\n"
    | when /def/
    |   print "includes /def/\n"
    | end
includes /def/
   nil
```

while

Ruby дает возможность с удобством создавать циклы, хотя, как вы узнаете из следующей главы, изучение методов использования *итераторов* ликвидирует необходимость в очень частом использовании циклов.

while - это как бы повторяющийся if. Мы использовали его в программе-пазле, угадываюшем слова и в программе (смотри предыдущую главу); там этот оператор выглядел как while *условие* ... end с блоком кода, выполнявшимся до тех пор, пока *условие* выполнялось (т.е. результатом выполнения условия было true). Но while и if также может быть легко применено и для отдельного выражения:

```
ruby> i = 0
   0
ruby> print "It's zero.\n" if i==0
It's zero.
   nil
ruby> print "It's negative.\n" if i<0
   nil
ruby> print "#{i+=1}\n" while i<3
1
2
3
    nil
```

Иногда желательно инвертировать результат выполнения проверки условия . unless - это обратное для if, а until - обратное для while. Мы оставляем Вам проверку того, как это работает. Поэкспериментируйте.

Существует 4 способа прервать выполнение цикла внутри него. Первый, break выполняет, как и в C, полный выход из цикла. Второй, next выполняет переход на начало следующей итерации (так же, как и continue в C). Третий - в Ruby есть оператор redo, который повторно выполняет текущую итерацию. Приведенный ниже код на C иллюстрирует действие операторов break, next, и redo:

```
while (condition) {
 label_redo:
     goto label_next          /* next */
     goto label_break         /* break */
     goto label_redo          /* redo */
     ;
     ;
 label_next:
}
label_break:
;
```

Четвертым способом выбраться из цикла является return. При выполнении return происходит не только выход из тела цикла, но также и возврат из метода, содержащего данный цикл. Если задан аргумент, то он будет его значение будет передано вызывающему методу, иначе возвращается nil.

for

Программистам на C уже, наверное, интересно, как же делаются циклы с "for". В Ruby for несколько более интересен, чем можно было бы предположить. Нижеприведенный цикл выполняется для каждого элемента из коллекции:

```
for elt in collection
   ...
```

```
end
```

Коллекция может быть представлена как диапазон значений (это как раз то, что подразумевает большинство людей, говоря о циклах "for"):

```
ruby> for num in (4..6)
    |    print num,"\n"
    | end
4
5
6
   4..6
```

Это также может быть некоторыми другими выдами коллекций, такими как массив:

```
ruby> for elt in [100,-9.6,"pickle"]
    |    print "#{elt}\t(#{elt.type})\n"
    | end
100     (Fixnum)
-9.6    (Float)
pickle (String)
   [100, -9.6, "pickle"]
```

Но мы збегаем несколько вперед. for, на самом деле, является другим способом записи each, который - так получилось - будет нашим первым примером использования итераторов. Две нижеприведенных примера являются эквивалентными:

```
#   If you're used to C or Java, you might prefer this.
for i in collection
   ...
end

#   A Smalltalk programmer might prefer this.
collection.each {|i|
   ...
}
```

Итераторы часто могут служить заменой для обычных циклов, и, однажды привыкнув их использовать, оказывается проще работать именно с ними. Итак, давайте двигаться вперед вперед - изучаем итераторы.

Back to top

Ruby -- руководство

Итераторы

пользователя

Итераторы не являются чем-то, свойственным только Ruby. Они общеупотребительны в объектно-ориентированных языках программирования. Они также есть в Lisp, хотя там они не называются итераторами. Как бы то ни было, концепция итераторов не является близкой множеству людей, так что давайте рассмотрим ее более детально.

Глагол *iterate* (повторять) означает "проделывать одно и то же много (ну, несколько) раз, так что итератор (*iterator*) - это нечто, делающее одно и то же много раз.

Когда мы программируем, то нуждаемся в использовании циклов во множестве различных ситуаций. В C мы кодируем их, используя for или while. Например,

```
char *str;
for (str = "abcdefg"; *str != '\0'; str++) {
   /* process a character here */
}
```

Синтакс оператора C for(...) обеспечивает абстракцию, помогающую в создании циклов, но для проверки *str на null необходимо знание программистом деталей о внутреннем представлении строки. Это дает представление о C как о низкоуровневом языке программирования. Высокоуровневые языки отличает более гибкая поддержка ими итераторов. Посмотрите на скрипт sh :

```
#!/bin/sh

for i in *.[ch]; do
   # ... этот код будет выполнен для каждой строки
done
```

Обрабатываются все исходники на C и хидеры в текущем каталоге , a shell заботится о деталях получения и подстановки имен файлов один за одним. Думаю, это работает на несколько более высоком уровне, чем C, а?

Можно покопать глубже: хотя и хорошо, что в языке есть поддержка для создания итераторов по встроенным типам данных, но разочаровывает, что когда необходима итерация по собственным типам данных, нужно возвращаться к низкоуровневым циклам. В ООП Пользователи часто создают один за другим собственные типы данных, так что такая ситуация может создать серьезные проблемы.

Так, каждый язык ООП включает в себя некоторые средства поддержки итераторов. В некоторых языках для этой цели существуют специальные классы; Ruby же позволяет определять итераторы напрямую.

В Ruby для типа String имеется несколько полезных итераторов:

```
ruby> "abc".each_byte{|c| printf "<%c>", c}; print "\n"
```

```
<a><b><c>
   nil
```

each_byte - итератор для каждого символа в строке. Каждый символ подставляется в локальную переменную c. Это может быть представлено как нечто, весьма напоминающее код на C...

```
ruby> s="abc";i=0
   0
ruby> while i<s.length
    |    printf "<%c>", s[i]; i+=1
    | end; print "\n"
<a><b><c>
   nil
```

... как бы то ни было, итератор each_byte и концептуально проще, и более вероятно продолжит работать даже если класс String будет в будущем радикально изменен. Одно из преимуществ итераторов - это то, что они имеют тенденцию выживать в случае подобных изменений; в самом деле, это является одной из характеристик хорошего кодирования вообще. (Ну да, потерпите немного, мы также собираемся поговорить о том, что такое *классы* вообще).

Вот еще итератор для String - each_line.

```
ruby> "a\nb\nc\n".each_line{|l| print l}
a
b
c
   nil
```

С задачами, которые потребовали бы больших усилий при программировании на C (поиск разделителей строк, выделение подстрок, и т.д.), легко справиться, используя итераторы.

Оператор for, использованный в предыдущей главе, делает итерации тем же путем, что и итератор each. each для String делает то же самое, что и each_line, так что давайте перепишем предыдущий пример с for:

```
ruby> for l in "a\nb\nc\n"
    |   print l
    | end
a
b
c
   nil
```

Мы можем использовать управляющую команду retry в сочетании с повторяющимся циклом, так что данная итерация будет повторена с самого начала.

```
ruby> c=0
   0
ruby> for i in 0..4
    |   print i
    |   if i == 2 and c == 0
    |     c = 1
    |     print "\n"
    |     retry
    |   end
    | end; print "\n"
012
01234
   nil
```

yield иногда присутствует в определении итератора. yield передает управление блоку кода, переданному в итератор (это будет более детально разъяснено в главе о процедурных объектах). В следующий примере определяется итератор repeat, который повторяет блок кода переданное в аргументе количество раз.

```
ruby> def repeat(num)
    |   while num > 0
    |     yield
    |     num -= 1
    |   end
    | end
   nil
ruby> repeat(3) { print "foo\n" }
foo
foo
foo
   nil
```

При помощи retry, можно определить итератор, который будет работать так же, как и while, хотя это слишком медленно, чтобы быть практичным.

```
ruby> def WHILE(cond)
    |   return if not cond
    |   yield
    |   retry
    | end
   nil
ruby> i=0; WHILE(i<3) { print i; i+=1 }
012   nil
```

Понятно, что такое итератор? Есть некоторые ограничения, но Вы можете написать собственные итераторы; фактически, когда Вы создаете новый тип данных, то часто

удобно определить подходящий для него итератор. В этом смысле приведенные выше примере не слишком полезны. Мы сможем говорить о практическом применении итераторов после того, как будем иметь лучшее представление о том, что же такое классы.

Back to top

"Объектно-ориентированный" - какое же это заразное выражение. Назовите что-то "объектно-ориентированным" - и Ваша фраза будет оччень умно. О Ruby говорят как об объектно-ориентированном языке; а что же точно означает "Объектно-ориентированный"?

Есть множество вариантов ответа на этот вопрос, но все они, вероятно, сводятся к одному и тому же. Чем слихком быстро их просуммировать, давайте немного поразмышляем о парадигме традиционного программирования

Традиционно задача в программировании сводится к некоторому виду *представления данных*, и *процедурам*, которые призводят операции над этими данными. В соответствии с этой моделью, данные являются инертными, пассивными и беспомощными; они полностью отданы на милость жирному телу процедур, которые активны, логичны и всемогущи.

Проблема, заключающаяся в таком подходе в том, что программы пишутся программистами, которые всего лишь люди и могут держать в голове ясными не так много деталей в данный момент. Как только проект вырастает, его процедурный слой растет до таких размеров, когда становится затруднительно помнить, как он работает в целом. Малейшие оплошности в мышлении и ошибки в наборе с большой долей вероятности приведут к хорошо упрятанным багам. Сложные и непредусмотренные взаимодействия начинают складываться в процедурном слое, его поддержка становится похожей на то, как если бы вы несли рассерженного спрута, пытаясь не допустить касания щупальцами Вашего лица. Существуют принципы программирования, которые могут помочь минимизировать и локализовать баги при традиционной парадигме, но есть лучшее решение, которое включает в себя фундаментальную смену принципов нашей работы.

Что нам дает объектно-ориентированное программирование - оно позволяет перенести повторяющуюся логику и механизмы взаимодействия на *сами данные*; оно меняет концепцию данных с *пассивной* на *активную*. Иначе говоря,

- Мы более не рассматриваем данные как некоторую коробку с откытой крышкой, которая позволяет нам заглянуть вовнутрь и пожевать содержимое.

- ы начинаем рассматривать данные как некий закрытый механизм с четко определенными рычажками и индикаторами.

То, что мы ранее назвали "устройством", может быть очень простым или сложным внутри; мы не можем определить это, глядя снаружи; также мы не позволяем себе заглянуть вовнутрь (за исключением случев, когда мы абсолютно уверены, что с дизайном что-то не так); таким образом, для взаимодействия от нас требуется лишь пощелкать переключателями и прочитать значения индикаторов. Когда этот механизм построен, нам не нужно думать как он работает внутри.

Можно подумать, что таким образом мы лишь усложняем себе работу, однако данный подход дает возможность предотвратить ситуации, когда что-то идет не так, как предполагалось.

Давайте нанем с примера; он слишком прост, чтобы быть практичным, однако способен, по крайней мере, частично проиллюстрировать саму концепцию. В Вашей машине есть одометр. Его работа заключается, среди прочего, в том, чтобы подсчитывать расстояние, пройденное с момента последнего нажатаи я его кнопки сброса. Как это можно смоделировать на языке программирования? На C одоиметр можно представить просто числовой переменной, возможно, с типом float. Такая программа будет работать с этой переменной увеличивая ее значение маленькими шагами, сбрасывая ее значение в ноль при необходимости. Что же здесь не так? Из-за ошибки в программе этой переменной может быть присвоено неверное значение просто по воле случая. Любой, кто программировал на C, знает как это часами или целыми днями выслеживал подобные ошибки, которые при обнаружении оказываются абсурдно элементарными. (Момент обнаружения бага обычно прослеживается по звуку шлепка по лбу.)

Решение той же самой проблемы в объектно-ориентированном (OO) контексте будет рассматриваться с совершенно другой стороны. Первым вопросом программиста при разработке модели одометра будет не "какой из существующих типов танных наиболее полно отражает порядок вещей", а "какой точно предполагается работа этой штуки". В этом и есть коренное различие. Необходимо потратить немного больше времени, обдумывая для чего же точно нужен одометр и как окружающий мир может с ним взамодействовать. Мы решаем построить небольшое устройство с управлением, позволяющем увеличить его показания, сбросить их, прочитать значение и ... ничего более...

Мы не даем возможности установить произвольное значение - почему? А потому что, как все мы знаем, одометр не работает таким образом. Есть только определенные действия, которые разрешено проводить с одометром. Таким образом, если что-то в программе пытается установить некоторое посторонне значение (скажем, целевую температуру системы климат-контроля машины) в одометр, немедленно станет ясно, что происходит что-то не то. Во время выполнения программы (или, возможно, во время компиляции, - зависит от вида языка) нам будет указано, что *недопустимо присваивать объекту Tripmeter произвольные значения*. Сообщение может быть не настолько ясным, но достаточно близким к этому. Это не предотвращает ошибки, не так ли? Но это быстро укажет нам направление поиска причины. Это только один из способов, которым OO программирование может предотвратить кучу впустую потраченного времени.

Мы обычно поднимаемся на уровень абстракции выше чем здесь, Потому как оказывается удобным построить factory, которая производит подобные устройства, чем создавать их поодиночке. Маловероятно, что мы прямо будем создавать объект одометр; вместо этого мы предпочтем, чтобы нужное их количество было построено по пределенному шаблону. Шаблон (или, если Вам больше нравится, tripmeter factory) соотносится с тем, что называют "*классом*", а отдельно взятый одометр соотносится с "*объектом*". В большинстве ОО языков требуется, чтобы класс был определен до того, как мы можем получить объект нового типа, но не в Ruby.

Здесь ничего не сказано о том, что использование ОО языка *вынуждает создавать* хороший ОО дизайн. В самом деле, на любом языке можно написать насквозь нечитаемую, неаккуратную, плохо продуманную, глючную и неустойчивую в работе программу. Что Ruby дает Вам (особо по сравнению с C++), так это то что, когда Вы прорабатываете детали, то не чувствуется желания ухудшить стиль для сохранения усилий. По ходу повествования мы будем обсуждать как же Ruby идет к этой прекрасной цели. Следующей темой будет "кнопочки и рычажки" (методы объектов), от которых мы перейдем к "фабрикам" (классам). Вы еще здесь?

Back to top

Что есть метод? В ОО программировании мы не думаем опрямом изменении данных объекта снаружи; вместо этого объекты сами знают о как с ними нужно работать (когда их об этом попросят в соответствующей форме.) Вы можете сказать, что мы шлем сообщения объекту, и эти сообщения в общем случае, вызывают некоторые действия или осмысленный отклик. Это должно происходить независимо от нашего представления или щаботы о том, как на самом деле действует объект. Действия, которые нам разрешено запрашивать у объекта (или, что равнозначно, сообщения, которые он понимает) являются *методами* объекта

В Ruby мы производим вызов метода, используя точечную нотацию (так же, как в C++ или Java). Имена объектов, чьи методы мы вызываем, располагаются справа от точки.

```
ruby> "abcdef".length
   6
```

Интуитивно, *этот строковый объект запрашивается о его длине.*. Технически, мы вызываем метод length объекта "abcdef".

Другие объекты могут иметь несколько иное представление о length, или вообще никакого. Решение, как отреагировать на сообщение, принимается на лету, во время

исполнения программы, и принимаемые действия могут меняться в зависимости от того, на что ссылается переменная.

```
ruby> foo = "abc"
   "abc"
ruby> foo.length
   3
ruby> foo = ["abcde", "fghij"]
   ["abcde", "fghij"]
ruby> foo.length
   2
```

Что мы имеем в виду под *длиной* может варьироваться в зависимости от того, о каком объекте мы говорим. В первый раз, когда мы запрашиваем foo о его длине в предыдущем примере, он ссылается на простую строку, и здесь может быть только один осмысленный ответ. Во второй раз foo ссылается на массив, и есть причины думать, что результатом будет 2, 5, or 10; но наиболее подходящим ответом будет, конечно, 2 (другие результаты также могут быть, по желанию, получены) .

```
ruby> foo[0].length
   5
ruby> foo[0].length + foo[1].length
   10
```

Необходимо отметить, что массив *что-то знает о том, что значит быть массивом*. Данные в Ruby несут такое знание, так что при необходимости запрос на них автоматически может быть удовлетворен несколькими способами. Это освобождает программиста от бремени запоминания множества специфичных имен функций, поскольку относительно небольшое число названий методов, соотносящихся с концепцией того, что мы знаем, как выразить на естественном языке, может быть применено к различным тиам данных, и результатом будет то, что мы предполагаем. Это свойство ОО языков программирования (которое, IMHO, Java использует не лучшим образом) называется *полиморфизм*.

Когда объект получает сообщение, которое он не понимает, "возбуждается" ошибка:

```
ruby> foo = 5
   5
ruby> foo.length
ERR: (eval):1: undefined method `length' for 5(Fixnum)
```

Таким образом, необходимо знать, какие методы принимаются объектом, несмотря на то, что нам не нужно знать, как они обрабатываются.

Если методу передаются аргументы, они обычно заключаются в скобки,

```
object.method(arg1, arg2)
```

но их можно опустить, если этим мы не вносим неоднозначность.

```
object.method arg1, arg2
```

В Ruby существует особая переменная self; она ссылается на вызывающий метод объект. Такое происходит настолько часто, что для удобства "self." может быть опущен, когда объект вызывает собственные методы:

```
self.method_name(args...)
```

есть то же самое, что и

```
method_name(args...)
```

Как можно подумать, *вызовы функций* являются просто укороченной записью вызова методов объекта self. Это делает Ruby тем, что называется чисто объектно-ориентированным языком. Все же функциональные методы ведут себя подобно функциям в других языках программирования для тех, кто не хочет понимать, что вызовы функций также на самом деле являются методами объектов Ruby. Мы можем говорить о *функциях* как если бы они не были методами объектов, если нам уж так этого хочется.

Back to top

Ruby -- руководство пользователя

Классы

Реальный мир полон объектов, и мы можем их классифицировать. Например, при виде собаки младенец, вероятно, скажет "bow-wow" независимо от породы; и мы в самом деле можем рассматривать окружающий мир в данных категориях.

В терминологии ОО программирования, категория объектов типа "собака" называется *класс*, а отдеьные объекты, принадлежащие классу, называются *представителями* этого класса.

В общем случае, перед созданием объекта в Ruby или любом другом ОО языке, мы сперва должны определит характеристики данного класса, а затем создавать представителей. Для иллюстрации процесса, давайте сперва для примера создадим класс Dog.

```
ruby> class Dog
```

```
    |   def speak
    |     print "Bow Wow\n"
    |   end
    | end
  nil
```

В Ruby определением класса заключаеься в ограничители class и end. def внутри этого раздела открывает определение *метода* класса, которое, как мы рассмотрели ранее, определяет некоторое специфическое поведение объектов данного класса.

Теперь, после определения класса Dog, мы можем использовать его для создания объекта "собака".

```
ruby> pochi = Dog.new
  #<Dog:0xbcb90>
```

Мы создали новый объект класса Dog, и дали ему имя pochi. Метод new любого класса создает объект данного класса. Поскольку pochi принадлежит классу Dog в соответствии с нашим определением класса он имеет все свойства, которые мы определили для класса Dog. Т.к. наша идея Dog-ности была очень простой, есть только одно действие, которое pochi можо попросить выполнить.

```
ruby> pochi.speak
Bow Wow
  nil
```

Создание нового объекта класса иногда называется *инстанциированием* этого класса. нам нужно получить собаку, прежде чем мы будем иметь удовольствие поговорить с ней; мы не можем просто попросить *class* Dog погавкать для нас.

```
ruby> Dog.speak
ERR: (eval):1: undefined method `speak' for Dog:class
```

В этом не больше смысла, чем пытаться *съесть концепцию бутерброда*.

С другой стороны, если мы хотим услышать лай собаки без привязки к чему-либо, мы можем создать (инстанциировать) эфемерную, временную собаку, и успеть извлечь из нее немного шума прежде чем она исчезнет.

```
ruby> (Dog.new).speak   # or more commonly, Dog.new.speak
Bow Wow
  nil
```

"Погодите," скажите Вы, "что там насчет несчастной, исчезающей в итоге?" Это правда - если мы не озаботимся дать ей имя (как мы сделали с pochi), автоматический сборщик мусора Ruby решит, что это ненужная бродячая собака и немилосердно избавится от

нее. На самом деле это хорошо, т.к. мы можем сделать себе столько собак, сколько нам надо.

Back to top

Наша классаификация объектов в повседеневной жизни естественно иерархична. Мы знаем, что *все кошки есть млекопитающие*, и *все млекопитающие есть животные*. Более мелкие классы *наследуют* характеристики более крупых, к которым они принадлежат.

Мы можем отразить эту концепцию в Ruby:

```
ruby> class Mammal
    |   def breathe
    |     print "inhale and exhale\n"
    |   end
    | end
   nil
ruby> class Cat<Mammal
    |   def speak
    |     print "Meow\n"
    |   end
    | end
   nil
```

Хотя мы явно не определили, как именно Cat должен дышать (breathe), каждый кот наследует поведение класса Mammal, поскольку Cat был определен как подкласс Mammal. (В ОО терминологии меньший класс является *подклассом* а больший (более общий) класс является *суперклассом*.) Следовательно, с точки зрения программиста, кошки (cats) бесплатно получают возможность дышать (breathe); после добавления метода speak method, наши кошки могут и дышать (breathe) и говорить (speak).

```
ruby> tama = Cat.new
   #<Cat:0xbd80e8>
ruby> tama.breathe
inhale and exhale
   nil
ruby> tama.speak
Meow
   nil
```

Бывают ситуации, когда некоторые свойства суперкласса не должны наследоваться определенным подклассом. Хотя обычно птицы знают как летать, пингвины являются нелетающим подклассом птиц.

```
ruby> class Bird
    |   def preen
    |     print "I am cleaning my feathers."
    |   end
    |   def fly
    |     print "I am flying."
    |   end
    | end
   nil
ruby> class Penguin<Bird
    |   def fly
    |     fail "Sorry. I'd rather swim."
    |   end
    | end
   nil
```

Вместо того, чтобы полностью определять все свойства каждого новото класса, мы можем просто добавить или переопределить их в соответствии с различиями между подклассом и суперклассом. Это использование наследования иногда называют *differential programming*. Это одно из преимуществ объектно-ориентированного программирования.

Back to top

Ruby -- руководство пользователя Переопределение методов

В подклассе мы можем переопределить поведение метода суперкласса его переопределением.

```
ruby> class Human
    |   def identify
    |     print "I'm a person.\n"
    |   end
    |   def train_toll(age)
    |     if age < 12
    |       print "Reduced fare.\n";
    |     else
```

```
     |       print "Normal fare.\n";
     |     end
     |   end
     | end
   nil
ruby> Human.new.identify
I'm a person.
   nil
ruby> class Student1<Human
     |   def identify
     |     print "I'm a student.\n"
     |   end
     | end
   nil
ruby> Student1.new.identify
I'm a student.
   nil
```

Предположим, мы хотим расширить метод identify суперкласса вместо его полной замены. . Для этого используется super.

```
ruby> class Student2<Human
     |   def identify
     |     super
     |     print "I'm a student too.\n"
     |   end
     | end
   nil
ruby> Student2.new.identify
I'm a human.
I'm a student too.
   nil
```

super позволяет передавать аргументы первоначальному методу. Иногда говорят, что существуют два типа людей...

```
ruby> class Dishonest<Human
     |   def train_toll(age)
     |     super(11) # we want a cheap fare.
     |   end
     | end
   nil
ruby> Dishonest.new.train_toll(25)
Reduced fare.
   nil

ruby> class Honest<Human
```

```
     |   def train_toll(age)
     |     super(age) # pass the argument we were given
     |   end
     | end
   nil
ruby> Honest.new.train_toll(25)
Normal fare.
   nil
```

Back to top

Ранее мы говорили, что в Ruby нет функций, только методы Тем не менее, есть более чем один вид методов. Earlier, we said that ruby has no functions, only methods. However there is more than one kind of method. В этой главе мы введем понятие *управления доступом*.

Посмотрим, что случится, если мы определим метод "top level", а не внутри определения некоторого класса. Можно представить данный метод аналогом *функции* в более традиционном языке программирования, таком как C.

```
ruby> def square(n)
     |   n * n
     | end
   nil
ruby> square(5)
   25
```

Кажется, что наш метод не принадлежит никакому классу, однако Ruby включает его в определение базового класса Object, который является суперклассом для любого другого класса. В результате любой объект в состоянии использовать данный метод. Все правильно, но есть некоторый подвох: это *private* (приватный) метод для любого класса. Ниже мы рассмотрим, что это означает, но одним из следствий является то, что он может быть вызван только как функция, как в примере:

```
ruby> class Foo
     |   def fourth_power_of (x)
     |     square(x) * square(x)
     |   end
     | end
   nil
ruby> Foo.new.fourth_power_of 10
```

```
10000
```

Мы не можем явно применить метод к объекту:

```
ruby> "fish".square(5)
ERR: (eval):1: private method `square' called for "fish":String
```

Это достаточно разумно для сохранения чистой ОО природы Ruby (функции - это все еще методы объектов, но получателем вызовов явно является self) при обеспечении возможности вызова функций, написанных на традиционных языках программирования.

Общепринятым подходом в ОО программирование, как мы обращали внимание ранее, является разделение *спецификации* и *реализации*, или *что* метод объекта должен выполнять и *как* он на внутри это делает. Внутренне поведение объекта в общем случае должно быть скрыто от его пользователей; их должно волновать лишь что они передают и что получают обратно, надеясь, что объект знает, как что делать внутри. Как таковое, часто бывает полезно иметь в классах методы, невидимые снаружи, но которые используются внутри (и могут быть изменены программистом при необходимости, без влияния на видение объектов данного класса конечными пользователями). В приведенном ниже тривиальном примере смотрите на engine как на невидимого внутреннего работягу.

```
ruby> class Test
    |   def times_two(a)
    |     print a," times two is ",engine(a),"\n"
    |   end
    |   def engine(b)
    |     b*2
    |   end
    |   private:engine  # this hides engine from users
    | end
   Test
ruby> test = Test.new
   #<Test:0x4017181c>
ruby> test.engine(6)
ERR: (eval):1: private method `engine' called for #<Test:0x4017181c>
ruby> test.times_two(6)
6 times two is 12.
   nil
```

Можно предположить, что test.engine(6) вернет значение 12, но, как мы знаем, engine недоступен пользователю объекта Test. только другие методы Test, такие как times_two, имеют право использовать engine. Нам необходимо использовать общедоступный интерфейс, который состоит из метода times_two. Программист, который разрабатывает этот класс, может свободно изменить engine (здесь, например, написав вместо b*2 строку b+b, предполагая большую производительность нового варианта) не воздействуя на взаимодействие пользователя с объектами класса Test objects. Безусловно, данный пример слишком прост чтобы быть полезным; преимущества управления доступом

становятся яснее только когда мы начинаем создавать более сложные и интересные классы.

Back to top

Ruby -- руководство пользователя

Singleton-методы

Поведение объекта класса определяется его принадлежностью к конкретному классу, но бывают случаи, когда некоторый объект должен иметь особое поведение. В большинстве языков мы должны в этом случае определить новый класс, который будет инстанциирован лишь один раз. В Ruby мы можем назначить любому объекту его собственные методы.

```
ruby> class SingletonTest
    |   def size
    |     print "25\n"
    |   end
    | end
   nil
ruby> test1 = SingletonTest.new
   #<SingletonTest:0xbc468>
ruby> test2 = SingletonTest.new
   #<SingletonTest:0xbae20>
ruby> def test2.size
    |   print "10\n"
    | end
   nil
ruby> test1.size
25
   nil
ruby> test2.size
10
   nil
```

В этом примере, test1 и test2 принадлежат одному и тому же классу, но для test2 переопределен метод size, так что он ведет себя по-другому. Метод, определенный только для единичного объекта, называется *singleton-методом*.

Singleton-методы часто используются для элементов (GUI), где при нажатии различных кнопок должны производиться различные действия.

Singleton-методы не являются характерными только для Ruby, они также присутствуют и в CLOS, Dylan, и т.д. Также в некоторых языках, например Self и NewtonScript, существуют

только singleton-методы. Их иногда называют *prototype-based* языки.

Back to top

Модули в Ruby аналогичны классам, за исключением:

- модуль не может иметь представителей.
- модуль не может иметь подклассов.
- модуль определяется конструкцией module ... end.

На самом деле... класс Module модуля является суперклассом класса Class класса. Понятно? Нет? Едем дальше.

Существуют два типа модулей. Один предназначен для централизованного хранения методов и констант. Модуль Math в стандартной библиотеке Ruby выступает в данной роли:

```
ruby> Math.sqrt(2)
   1.41421
ruby> Math::PI
   3.14159
```

Оператор :: указывает интерпретатору Ruby местоположение константы (возможно, некоторые модули кроме Math определяют значение PI несколько иначе). Если мы хотим прямо ссылаться на методы или константы модуля без использования ::, мы можем включить (include) нужный модуль:

```
ruby> include Math
   Object
ruby> sqrt(2)
   1.41421
ruby> PI
   3.14159
```

Другим способом использования модулей является миксин (*mixin*). Некоторые ОО языки программирования, включая C++, допускают *множественное наследование*, то есть наследование от более чем одного суперкласса. Примером множественного наследования из реальной жизни может служить будильник; можно считать, что будильник принадлежит к классу *часы* и также к классу *жужжалка*.

Ruby умышленно не разрешает множественное наследование, но техника *mixin-ов* является хорошей альтернативой. Помните, что модули не могут быть инстанциированы и от них нельзя образовать подклассы; но если мы включаем (include) модуль в определение класса, его методы добавляются, или подмешиваются ("mixed in") в класс.

Миксины можно рассматривать с позиций того, какие частные свойства мы хотим получить. Например, если у класса есть работающий метод each, включая в него модуль Enumerable из стандартной библиотеки бесплатно дает нам методы sort и find.

Подобное использование модулей дает нам простейшую функциональность множественного наследования, но дает возможность представить взаимоотношения классов простой структурой дерева, а также значительно упрощает реализацию языка (подобный выбор также был сделан дизайнерами Java).

Back to top

Ruby -- руководство пользователя

Процедурные объекты

Часто необходимо обеспечить реакцию системы на нестандартное событие. Когда оно происходит, проще всего это сделать если мы можем передать блок кода в качестве аргумента другому методу, что означает, что мы хотим интерпретировать код как если бы это были данные.

Новый *процедурный объект* создается используя proc:

```
ruby> quux = proc {
    |   print "QUUXQUUXQUUX!!!\n"
    | }
   #<Proc:0x4017357c>
```

Здесь то, на что ссылается quux является объектом, и, как большинство объектов, имеет некоторое поведение, которое может быть запущено. В частности, мы можем попросить его выполниться через его метод call:

```
ruby> quux.call
QUUXQUUXQUUX!!!
   nil
```

Итак, в итоге, может ли quux быть использован в качестве аргумента? Конечно.

```
ruby> def run( p )
```

```
      |   print "About to call a procedure...\n"
      |   p.call
      |   print "There: finished.\n"
      | end
    nil
ruby> run quux
About to call a procedure...
QUUXQUUXQUUX!!!
There: finished.
    nil
```

Метод trap дает возможность задать реакцию по нашему выбору на любой системный сигнал.

```
ruby> inthandler = proc{ print "^C was pressed.\n" }
    #<Proc:0x401730a4>
ruby> trap "SIGINT", inthandler
    #<Proc:0x401735e0>
```

В нормальных условиях нажатие *^C* заставляет интерпретатор завершить выполнение. Теперь же печатается сообщение и интерпретатор продолжает выполнение, так что вы не теряете выполняемую работу. (Но вы не застряли в нем вечно - можно завершить его набрав exit или нажав *^D*.)

Последнее замечание перед переходом к другой теме: нет необходимости назначать процедурному объекту имя перед привязкой его к сигналу. Эквивалентный *анонимный* процедурный объект будет выглядеть как:

```
ruby> trap "SIGINT", proc{ print "^C was pressed.\n" }
    nil
```

или, еще более компактно,

```
ruby> trap "SIGINT", 'print "^C was pressed.\n"'
    nil
```

Краткая форма записи дает некоторое удобство и читабельность, когда Вы пишете маленькие анонимные процедуры.

Back to top

Ruby -- руководство пользователя

Переменные

В Ruby существуют три вид переменных, один вид констант и точно две псевдо-переменные. Переменные и константы не имеют типа. Хотя у нетипизированных переменных есть некоторые недостатки, они имеют больше преимуществ и хорошо вписываются в философию Ruby *быстро и просто*.

В большинстве языков переменные должны быть объявлены для указания их типа, изменяемости (т.е. являются ли они константами), и облати действия; Поскольку тип не задается, а остальное видно из имени переменной, объявление переменных в Ruby не нужно.

Первая буква идентификатора дает нам возможность навскидку определить категорию:

$	глобальная переменная
@	переменная в классе
[a-z] or _	локальная переменная
[A-Z]	константа

Единственным исключением являются псевдо-переменные в Ruby: self, который всегда ссылается на выполняемый в данное время объект, и nil, неопределенное значение, присваиваемое неинициализированным переменным. У обоих имена, подходящие под категорию локальных, но self является глобальной переменной, поддерживаемой интерпретатором, а nil на самом дле константа. Так как есть только два исключения, они не портят картину в целом.

Вы не можете присвоить значение self или nil.main, поскольку self ссылается на объект верхнего уровня:

```
ruby> self
   main
ruby> nil
   nil
```

Back to top

Имя глобальной переменной начинается с $. оно может быть использовано в любом месте программы. До инициализации глобальная переменная содержит специальное значение nil.

```
ruby> $foo
   nil
```

```
ruby> $foo = 5
   5
ruby> $foo
   5
```

Необходимо соблюдать умеренность в использовании глобальных переменных. они опасны, поскольку их значение может быть изменено отовсюду. Чрезмерное их использование может сделать трудной локализацию ошибок; это также является показателем непродуманности дизайна Вашей программы. Если уж без использования глобальной переменной не удается обойтись, убедитесь в том, что Вы даете ей самоописательное имя, вероятность повторного неумышленного использования которого в дальнейшем низка (присвоение глобальной переменной имени типа $foo вероятно будет плохой идеей).

Одним хорошим свойством глобальной переменной является то, что его можно трассировать; Вы можете определить процедуру, которая будет вызываться при изменении значения переменной.

```
ruby> trace_var :$x, proc{print "$x is now ", $x, "\n"}
   nil
ruby> $x = 5
$x is now 5
   5
```

Когда глобальную переменную заставляют работать в качестве триггера для вызова процедуры при ее изменении, ее иногда называют *активной переменной*. Например, это полезно для поддержание своевременного обновления в GUI.

Ниже приведена подборка особых переменных, чье имя состоит из знака доллара ($) с последующим одним символом. Например, $$ содержит id процесса интерпретатора Ruby, является read-only. Здесь приведены основные системные переменные и их значение (смотрите детали в справочном руководстве по Ruby):

$!	последнее сообщение об ошибке
$@	местоположение ошибки
$_	последняя строка, прочитанная gets
$.	*номер* последней строки, прочитанной интерпретатором
$&	последняя строка, обработанная regexp
$~	последнее совпадение с regexp, как массив подвыражений
$*n*	*nth*-е подвыражение в последнем совпадении(то же самое, что и $~[*n*])
$=	флаг зависимости от регистра символов
$/	разделитель во входных строках
$\	разделитель в выходных строках
$0	имя файла со скриптом Ruby
$*	аргументы командной строки

$$	ID процесса интерпретатора
$?	статус завершения последнего дочернего процесса

Выше $_ и $~ имеют локальную область действия. Судя по именам, они должны быть глобальными, но в таком виде они более полезны и есть исторические причины для использования этих имен.

Back to top

Ruby -- руководство пользователя

Переменные экземпляра класса

Переменные экземпляра класса имеют имена, начинающиеся с @, и область их действия ограничена тем, на что можно сослаться, используя self. Два разных объекта, даже если они являются объектами одного и того же класса, могут иметь разные значения в своих переменные экземпляра класса (@-переменные). Снаружи объекта его @-переменные не могут быть не только изменены, но даже и просмотрены (т.е. в Ruby @-переменные никогда не являются *public*), за исключением случаев, когда это явно определено программистом. Как и в случае глобальных переменных, @-переменные содержат заначение nil если они не были инициализированы.

@-переменные в Ruby не нуждаются в декларировании. Это подразумевает гибкость структуры объекта. Фактически, каждая @-переменная динамически добавляется в объект при первой ссылке на нее.

```
ruby> class InstTest
    |   def set_foo(n)
    |     @foo = n
    |   end
    |   def set_bar(n)
    |     @bar = n
    |   end
    | end
   nil
ruby> i = InstTest.new
   #<InstTest:0x83678>
ruby> i.set_foo(2)
   2
ruby> i
   #<InstTest:0x83678 @foo=2>
ruby> i.set_bar(4)
   4
ruby> i
```

```
   #<InstTest:0x83678 @foo=2, @bar=4>
```

Обратите внимание, что i не сообщает о значении @bar пока не выполнен метод set_bar.

Back to top

Имя локальной переменной должно начинаться с латинской буквы в нижнем регистре или символа подчеркивания (_). Локальная переменная, в отличие от глобальной или @-переменной не содержит значение nil до инициализации:

```
ruby> $foo
   nil
ruby> @foo
   nil
ruby> foo
ERR: (eval):1: undefined local variable or method `foo' for main(Object)
```

Первое присваивание значения локальной переменной играет роль ее объявления. Если вы ссылаетесь на неинициализированную локальную переменную, интерпретатор Ruby считает, что вы пытаетесь вызвать метод с этим именем; сообщение об ошибке вы уже видели ранее.

В общем случае, область видимости локальной переменной одна из:

- proc{ ... }
- loop{ ... }
- def ... end
- class ... end
- module ... end
- тело программы (если ни один из приведенных выше случаев не подходит)

В следующем примере defined? - это оператор, проверяющий был ли определен идентификатор. Он возвращает описание идентификатора, если он был определен, или nil в противном случае. Как вы видите, область действия bar локализована в цикле; при завершении цикла bar становится неопределен.

```
ruby> foo = 44; print foo, "\n"; defined? foo
44
   "local-variable"
ruby> loop{bar=45; print bar, "\n"; break}; defined? bar
45
```

```
   nil
```

Процедурные объекты, живущие в той же области действия, что и локальные переменные, делят общие локальные переменные, принадлежащие это области. Итак, локальная переменная bar принадлежит main и процедурным объектам p1 и p2:

```
ruby> bar=0
   0
ruby> p1 = proc{|n| bar=n}
   #<Proc:0x8deb0>
ruby> p2 = proc{bar}
   #<Proc:0x8dce8>
ruby> p1.call(5)
   5
ruby> bar
   5
ruby> p2.call
   5
```

Заметьте, что "bar=0" вначале не может быть опущена; это присваивание гарантирует, что bar охватывается в p1 и p2. Иначе p1 и p2 определят свои собственные локадьные переменные bar, и вызов p2 приведет к ошибке "undefined local variable or method".

Мощь процедурных объектов следует из их способности быть переданными в качестве аргументов: разделяемые локальные переменные остаются действующими даже если они передаются из своей наальной области действия.

```
ruby> def box
    |   contents = 15
    |   get = proc{contents}
    |   set = proc{|n| contents = n}
    |   return get, set
    | end
   nil
ruby> reader, writer = box
   [#<Proc:0x40170fc0>, #<Proc:0x40170fac>]
ruby> reader.call
   15
ruby> writer.call(2)
   2
ruby> reader.call
   2
```

Ruby особо сообразителен в том, что касается области действия. Из наших примеров очевидно, что переменная contents разделяется между reader и writer. Но мы можем создать множественные пары reader-writer используя box как определено выше; каждая пара делит общую переменную contents, и эти пары не пересекаются друг с другом.

```
ruby> reader_1, writer_1 = box
   [#<Proc:0x40172820>, #<Proc:0x4017280c>]
ruby> reader_2, writer_2 = box
   [#<Proc:0x40172668>, #<Proc:0x40172654>]
ruby> writer_1.call(99)
   99
ruby> reader_1.call
   99
ruby> reader_2.call
   15
```

Back to top

Ruby -- руководство пользователя

Константы класса

Имя константы начинается с латинской буквы верхнего регистра. значение ей должно присваиваться максимум один раз. В текущей реализации Ruby повторное присваивание значения константе вызывает предупреждение, но не ошибку (не-ANSI версия eval.rb не показывает предупреждения):

```
ruby>fluid=30
   30
ruby>fluid=31
   31
ruby>Solid=32
   32
ruby>Solid=33
   (eval):1: warning: already initialized constant Solid
   33
```

Константы могут быть определены внутри классов, но в отличие от @-переменных, они доступны за пределами класса.

```
ruby> class ConstClass
    |   C1=101
    |   C2=102
    |   C3=103
    |   def show
    |     print C1," ",C2," ",C3,"\n"
    |   end
    | end
```

```
   nil
ruby> C1
ERR: (eval):1: uninitialized constant C1
ruby> ConstClass::C1
   101
ruby> ConstClass.new.show
101 102 103
   nil
```

Константы также могут быть определены в модулях.

```
ruby> module ConstModule
    |   C1=101
    |   C2=102
    |   C3=103
    |   def showConstants
    |     print C1," ",C2," ",C3,"\n"
    |   end
    | end
   nil
ruby> C1
ERR: (eval):1: uninitialized constant C1
ruby> include ConstModule
   Object
ruby> C1
   101
ruby> showConstants
101 102 103
   nil
ruby> C1=99  # not really a good idea
   99
ruby> C1
   99
ruby> ConstModule::C1  # the module's constant is undisturbed ...
   101
ruby> ConstModule::C1=99
ERR: (eval):1: compile error
(eval):1: parse error
ConstModule::C1=99
               ^
ruby> ConstModule::C1  # ... regardless of how we tamper with it.
   101
```

Back to top

При выполнении программы могут возникать непредвиденные проблемы. Файл, который нужно прочитать, может не существовать; диск может оказаться полным в момент записи данных; пользователь может некорректно ввести данные.

```
ruby> file = open("some_file")
ERR: (eval):1:in `open': No such file or directory - some_file
```

"Устойчивая" программа должна четко и изящно обработать такую ситуацию. Предусмотреть подобное может быть мучительной, изматывающей задачей. Предполагается, что программисты на C должны проверять результат каждого системного вызова, который потенциально может завершиться неудачей, и немедленно решить что должно быть сделано в этом случае:

```
FILE *file = fopen("some_file", "r");
if (file == NULL) {
   fprintf( stderr, "File doesn't exist.\n" );
   exit(1);
}
bytes_read = fread( buf, 1, bytes_desired, file );
if (bytes_read != bytes_desired ) {
   /* do more error handling here ... */
}
...
```

Это настолько утомительно, что у программиста может наблюдаться тенденция к росту небрежности; в результате чего программа не обрабатывает надежно исключения. С другой стороны, правильное выполнение работы делает текст программы нецдобочитаемым, поскольку настолько громоздкая обработка ошибок замусоривает значащий код.

В Ruby, как в большинстве современных языков, мы обрабатываем исключения для блоков кода раздельно, таким образом работая эффективно с подобными сюрпризами и не перегружая напрасно программиста или того человека, который позднее будет читать этот код. Блок кода, помеченный begin до тех пор, пока не происходит исключительная ситуация, которая передает управление блоку, отвечающему за ее обработку (начинается с rescue). Если исключения не происходит, то rescue-блок не выполняется. Следующий метод возвращает первую строку из текстового файла или nil, если происходит исключение:

```
def first_line( filename )
```

```
    begin
      file = open("some_file")
      info = file.gets
      file.close
      info    # Last thing evaluated is the return value
    rescue
      nil     # Can't read the file? then don't return a string
    end
  end
```

Иногда бывает нужна возможность обойтись с проблемой творчески. Здесь, если требуемый файл недоступен, мы пробуем использовать стандартный поток ввода:

```
  begin
    file = open("some_file")
  rescue
    file = STDIN
  end

  begin
    # ... process the input ...
  rescue
    # ... and deal with any other exceptions here.
  end
```

retry может быть использован внутри rescue для запуска повторного выполнения блока с begin. Теперь можно переписать предыдущий пример немного компактнее:

```
  fname = "some_file"
  begin
    file = open(fname)
    # ... process the input ...
  rescue
    fname = "STDIN"
    retry
  end
```

Тем не менее, здесь есть недостаток. Несуществующий файл может заставить этот код выполняться снова и снова. Вы должны остерегаться подобной западни при использовании retry для обработки исключительных ситуаций.

Любая библиотека Ruby возбуждает исключения при возникновении любой ошибки, и вы также можете явно возбуждать ошибки в своем коде. Для этого используйте raise. Этот оператор принимает один аргумент, который представляет из себя строку, описывающую исключение. Аргумент не обязателен, но опускать его не стоит. Его можно получить позже из специальной глобальной переменной $!.

```
ruby> raise "test error"
   test error
ruby> begin
    |   raise "test2"
    | rescue
    |   print "An error occurred: ",$!, "\n"
    | end
An error occurred: test2
   nil
```

Back to top

Ruby -- руководство пользоватуля

Обработка исключительных ситуаций: ensure

Иногда бывает нужна очистка по завершении работы некоторого метода. Возможно, должен быть закрыт открытый файл, буферизованные данные должны быть сброшены, и т.п., etc. Если бы существовала только одна точка выхода из метода, мы могли бы уверенно поместить код очистки в одно место и быть уверены, что он будет выполнен; однако возврат из метода может происходить из нескольких точек, или наша очистка может быть пропущена из-за возникновения исключения.

```
begin
  file = open("/tmp/some_file", "w")
  # ... write to the file ...
  file.close
end
```

В этом примере, если ошибка возникает во время записи файла, файл останется открытым. Также не хочется обращаться к подобной избыточности:

```
begin
  file = open("/tmp/some_file", "w")
  # ... write to the file ...
  file.close
rescue
  file.close
  fail # raise an exception
end
```

Это неуклюже, к тому же становится неуправляемым при усложнении кода, поскольку необходимо реагировать на каждые return и break.

По этой причине вводится новое ключевое слово в схему "begin...rescue...end" -- ensure. Блок ensure выполняется независимо от успешности выполнения блока begin.

```
begin
  file = open("/tmp/some_file", "w")
  # ... write to the file ...
rescue
  # ... handle the exceptions ...
ensure
  file.close      # ... and this always happens.
end
```

Возможно использование ensure без rescue, или наоборот, но если они использованы вместе в одном блоке begin...end , rescue должен предшествовать ensure.

Back to top

Ruby -- руководство пользователя

Аксессоры

Что такое аксессор?

Мы кратко обсудили @-переменные в предшествующих главах, но мало поэкспериментировали с ними. @-переменные объекта являются его атрибутами, тем, что обычно отличает его от других объектов этого же класса. Важно иметь возможность задавать и считывать значения атрибутов; для этого необходимо написание методов, называемых *аксессорами атрибутов*. Через мгновение мы увидим, что не всегда нужно явное задание аксессоров, но давайте сейчас полностью пройдем по этому пути. Есть два вида аксессоров - *writers* и *readers*.

```
ruby> class Fruit
    |   def set_kind(k)  # a writer
    |     @kind = k
    |   end
    |   def get_kind     # a reader
    |     @kind
    |   end
    | end
   nil
ruby> f1 = Fruit.new
   #<Fruit:0xfd7e7c8c>
ruby> f1.set_kind("peach")  # use the writer
   "peach"
```

```
ruby> f1.get_kind           # use the reader
   "peach"
ruby> f1                    # inspect the object
   #<Fruit:0xfd7e7c8c @kind="peach">
```

Достаточно просто; мы можем сохранять и считывать информацию о том, на фрукт какого вида мы смотрим. Но имена наших методов слегка многословны. Пример далее имеет более ясный и привычный вид:

```
ruby> class Fruit
    |   def kind=(k)
    |     @kind = k
    |   end
    |   def kind
    |     @kind
    |   end
    | end
   nil
ruby> f2 = Fruit.new
   #<Fruit:0xfd7e7c8c>
ruby> f2.kind = "banana"
   "banana"
ruby> f2.kind
   "banana"
```

Метод inspect

Краткое отступление. Вы уже заметили, что когда мы прямо рассматриваем объект, нам показывают нечто загадочное, типа #<anObject:0x83678>. Это просто действие по умолчанию, и мы свободно можем изменить его. Все, что нужно сделать - это добавить метод inspect в определение класса. Он должен возвращать строковое значение, описывающее объект подходящим способом, включая состояние всех или некоторых @-переменные.

```
ruby> class Fruit
    |   def inspect
    |     "a fruit of the " + @kind + " variety"
    |   end
    | end
   nil
ruby> f2
   "a fruit of the banana variety"
```

Ему соответствует метод to_s (перевести в строку), который используется для вывода на печать объекта. В общем случае, можно представлять себе inspect как средство, используемое при написании и отладке программ, а to_s как способ обогащения вывоа

программы. eval.rb использует inspect для вывода результатов. Вы можете использовать метод p для выполнения простого вывода при отладке программы.

```
# Эти две строки эквивалентны:
p anObject
print anObject.inspect, "\n"
```

Легкий способ создания аксессоров

Поскольку многие @-переменные нуждаются в применении аксессоров, Ruby предоставляет удобные сокращения для стандартных форм.

Сокращение	*Эффект (значение)*
attr_reader :v	def v; @v; end
attr_writer :v	def v=(value); @v=value; end
attr_accessor :v	attr_reader :v; attr_writer :v
attr_accessor :v, :w	attr_accessor :v; attr_accessor :w

Давайте воспользуемся этими преимуществами и добавим в пример информацию о свежести. Сперва мы затребуем автоматическую генерацию reader-а и writer-а, а затем включим новую информацию в inspect:

```
ruby> class Fruit
    |   attr_accessor :condition
    |   def inspect
    |     "a " + @condition + @kind"
    |   end
    | end
   nil
ruby> f2.condition = "ripe"
   "ripe"
ruby> f2
   "a ripe banana"
```

More fun with fruit

Никто не ест гнилые фрукты - так пусть время делает свое черное дело.

```
ruby> class Fruit
    |   def time_passes
    |     @condition = "rotting"
    |   end
    | end
   nil
ruby> f2
   "a ripe banana"
```

```
ruby> f2.time_passes
   "rotting"
ruby> f2
   "a rotting banana"
```

Но играя с нашим классом, мы внесли некоторую проблему. Что будет, если мы попытаемся сейчас создать третий фрукт? Вспомните, @-переменные не существуют до тех пор, пока им не присваивается какое-либо значение.

```
ruby> f3 = Fruit.new
ERR: failed to convert nil into String
```

Это метод inspect жалуется здесь, и на то есть причина. Мы попросили его дать отчет о типе и состоянии фрукта, но до сих пор f3 не был присвоен не один атрибут. Если бы мы пожелали, то можно было бы переписать метод inspect таким образом, чтобы он проверял определенность @-переменных используя метод defined? и давал по ним отчет только при положительном исходе, но, наверное, так будет не учень хорошо; т.к. каждый фрукт имеет вид и состояние, то желательно было бы каким-либо образом обеспечить их задание. Это и будет темой следующей главы.

Back to top

Ruby -- руководство пользователя

Инициализация объектов

В нашем классе Fruit из предыдущей главы есть @-переменные: первая описывает вид фрукта, вторая - его состояние. Только после написания своего специализированного метода inspect для этого класса, мы осознали, что для любого фрукта отсутствие заданных характеристик бессмысленно. К счастью, в Ruby есть способ гарантировать, что @-переменные всегда будут инициализированы.

Метод initialize

Когда Ruby создает новый объект, он ищет в описании класса метод initialize и вызывает его. Таким образов, простая вещь, которую мы можем проделать,- это использовать initialize для задания всем @-переменным значений "по-умолчанию", так что теперь "методу inspect всегда есть что сказать". ;-)

```
ruby> class Fruit
    |   def initialize
    |     @kind = "apple"
    |     @condition = "ripe"
    |   end
```

```
    | end
   nil
ruby> f4 = Fruit.new
   "a ripe apple"
```

От предположения к требованию

Бывает, что задавать значения "по-умолчанию" бессмысленно. Существует ли в природе такая вещь, как "фрукт по умолчанию"? наверное, будет предпочтительно, чтобы вид каждого фрукта задавался во время его создания. для этого необходимо ввести формальный аргумент в метод initialize . Мы не будем здесь вдаваться в причину почему, но аргументы, которые мы задаем в вызове метода new на самом деле передаются в метод initialize.

```
ruby> class Fruit
    |   def initialize( k )
    |     @kind = k
    |     @condition = "ripe"
    |   end
    | end
   nil
ruby> f5 = Fruit.new "mango"
   "a ripe mango"
ruby> f6 = Fruit.new
ERR: (eval):1:in `initialize': wrong # of arguments(0 for 1)
```

Гибкая инициализация

Как видно из примера выше, если с методом initialize ассоциирован аргумент, то его при создании объекта его нельзя отбросить с тем, чтобы не получить сообщения об ошибке. Если мы хотим быть деликатнее, то можем использовать аргумент, если он задан, в противном случае откатиться на его значение "по-умолчанию".

```
ruby> class Fruit
    |   def initialize( k="apple" )
    |     @kind = k
    |     @condition = "ripe"
    |   end
    | end
   nil
ruby> f5 = Fruit.new "mango"
   "a ripe mango"
ruby> f6 = Fruit.new
   "a ripe apple"
```

Вы можете использовать значения "по-умолчанию" для любых методов, а не только для initialize. Список аргументов должен быть построен так, чтобы аргументы, которые могут

иметь значения "по-умолчанию", шли последними.

Иногда полезно дать возможность инициализировать объект несколькими способами. Хотя это и выходит за рамки данного руководства, Ruby поддерживает reflection объектов и список аргументов переменной длины, которые вместе дают возможность эффективной перезагрузки методов.

Back to top

Эта глава посвящена некоторым практическим приемам.

Разделители выражений

В некоторых языках необходимо соблюдение некоторых правил пунктуации; часто используется точка с запятой (;), для обозначения окончания выражения в программе. Ruby вместо этого следует соглашениям, используемым в шеллах, таких как sh или csh. Множественные выражения в одной строке должны разделяться точкой с запятой, но это не обязательно в конце строки; перевод строки трактуется как точка с запятой. Если линия заканчивается обратным слэшем (\), перевод строки игнорируется; это позволяет разнести длинный оператор на несколько строк.

Комментарии

Зачем писать комментарии? Хотя хорошо написанный код имеет тенденцию к самодокументированию, часто бывает полезно делать заметки на полях; будет ошибкой думать, что другие смогут немедленно сказать что делает Ваш код лишь мельком взглянув на него (как это можете Вы). Кроме того, вы можете оказаться в роли этого другого человека всего лишь несколько дней спустя; кто из нас не возвращался спустя некоторое время к старой программе чтобы что-то дописать или исправить ее и не говорил "Я знаю, что это писал я. Но, черт побери, что вот это значит?"

Некоторые опытные программисты достаточно правильно заметят, что противоречивые и устаревшие комментарии могут быть хуже, чем их отсутствие. Безусловно, комментарии не могут быть заменой для читабельного кода; если Ваш код неясен, вероятно, он также содержит ошибки. Может оказаться, что, пока Вы изучаете Ruby, Вы больше испытываете нужду в комментариях, а затем все меньше и меньше, по мере того, как Вы учитесь все лучше выражать свои идеи с помощью простого, элегантного, читабельного кода.

Ruby следует общепринятому в скриптах правилу, по которому начало комментария выделяется знаком фунта (#). Все, что следует за ним, интерпретатором игнорируется.

Также, для облегчения написания больших комментариев, интерпретатор Ruby также игнорирует все, что заключено между строкой, начинающейся с "=begin" и заканчивающейся на "=end".

```
#!/usr/local/bin/ruby

=begin
**********************************************************************
   This is a comment block, something you write for the benefit of
   human readers (including yourself).  The interpreter ignores it.
   There is no need for a '#' at the start of every line.
**********************************************************************
=end
```

Упорядочиваем код

Интерпретатор Ruby обрабатывает код по мере его чтения. Нет ничего похожего на фазу компиляции; если что-то еще не было прочитано, значит, это просто неопределено.

```
# this results in an "undefined method" error:

print successor(3),"\n"

def successor(x)
   x + 1
end
```

Это не заставляет Вас, как может показаться на первый взгляд, располагать код в стиле "сверху-вниз". Когда интерпретатор натыкается на определение метода, он может благополучно проглотить неопределенные ссылки, но к тому времени, когда метод будет вызван, все они должны быть определены:

```
# Conversion of fahrenheit to celsius, broken
# down into two steps.

def f_to_c(f)
   scale (f - 32.0)    # This is a forward reference, but it's okay.
end

def scale(x)
   x * 5.0 / 9.0
end

printf "%.1f is a comfortable temperature.\n", f_to_c( 72.3 )
```

Хотя это может показаться менее удобным, чем то, к чему Вы, может быть, привыкли в Perl или Java, это накладывает меньшие ограничения, чем программирование на C без прототипов (что вынуждает Вас постоянно поддерживать частичную упорядоченность

ссылок). Размещение кода верхнего уровня внизу исходника работает всегда. Но даже это не так досадно, как может сперва показаться. Разумный и безболезненный способ обеспечить нужное Вам поведение - определить функцию main в начале файла, а затем вызвать ее снизу.

```
#!/usr/local/bin/ruby

def main
  # Put your top level code here...
end

# ... put all your support code here, organized as you see fit ...

main # ... and start execution here.
```

Также может помочь то, что в Ruby есть утилиты, разбивающие сложную программу на читабельные, с возможностью повторного использования, логически взаимосвязанные куски. Мы уже встречались с использованием include для доступа к модулям. Вы также сочтете полезным использование средств load и require. load работает как если бы файл, на который он ссылается, был скопирован и помещен в текст программы вместо него (несто похожее на директиву препроцессора #include в C). require - несколько более сложная вещь, код загружается только один раз и только тогда, когда он реально нужен. Есть и другие отличия между load и require; их можно посмотреть в справочном руководстве или в FAQ.

Се ля ви...

Этого тьюториала должно быть достаточно, чтобы помочь вам начать писать программы на Ruby. Если возникают вопросы, Вы всегда можете порыться в справочном руководстве чтобы "углубить и расширить" ;-) свои познания . FAQ и справочник по библиотеке также являются важными ресурсами.

Удачи, и счастливого кодирования!

Back to top

Одностраничная версия руководства пользователя Ruby
Fri Feb 16 13:56:18 CST 2001 (Eng)
Mon Sep 10 16:16:48 CST 2001 (Rus)