Ruby - новые грани

Автор: ***Евгений Охотников***
http://www.intervale.ru
Источник: ***RSDN Magazine #4-2006***

Опубликовано: *03.03.2007*
Исправлено: *12.07.2007*
Версия текста: *1.0*

1 Введение

Когда некоторое время назад я выбрал Ruby для реализации одной конкретной и не очень сложной задачи, то мне не удалось оценить основные грани Ruby в полной мере. Блоки кода. Необязательные скобки. Удобство использования attr_reader/attr_accessor. Наличие method_missing. Пожалуй, и все.

Вначале я программировал на Ruby в стиле C++, только пользуясь Ruby-новыми конструкциями. Более того, временами это было «через силу», мне очень не хватало статической типизации — диссонанс от того, что на новый язык переносилась философия старого языка.

А потом понемногу, по чуть-чуть, я оценил остальные особенности Ruby — как раз те, которые я описываю здесь. Мне потребовалось около девяти месяцев на то, чтобы разглядеть действительно наиболее яркие грани языка. А уже после того, как они четко обозначились, я стал решать задачи на Ruby, думая уже в терминах Ruby, а не C++.

Возможно, столь много времени потребовалось еще из-за того, что Programming Ruby [2] — это очень большая книга, которая посвящает читателя во все тонкости языка, но не расставляет акцентов. Однако при переходе с C++ на Ruby (как в моем случае) не столь важны конкретные особенности синтаксиса языка. Гораздо важнее осознать, как именно должно измениться мышление, чтобы при программировании на Ruby думать именно на Ruby, а не на C++. И, по-моему, мне еще не попадалась книга о Ruby, в которой делался бы акцент именно на этом.

Поэтому, когда представилась возможность рассказать о Ruby то, что я сам хотел бы рассказать, я решил описать те грани языка, на которых я сам в конце концов сосредоточился, и которые мне лично кажутся самыми важными. Соответственно, излагаемый ниже материал рассчитан на более-менее опытных программистов, уже использующих другие языки программирования, но желающих понять, что же в этом Ruby такого особенного. А также для тех, кому вольно или невольно приходится изучать Ruby — в качестве еще одного русскоязычного источника информации о Ruby. Несколькими словами данную статью можно охарактеризовать как «глубокое погружение в Ruby для тех, кто не прочитал Programming Ruby».

2 Язык Ruby вчера и сегодня

История языка Ruby началась в 1993 году, когда Якихиро Мацумото (Yukihiro Matsumoto) a.k.a Matz взялся за реализацию собственного скриптового языка, который бы был таким же мощным и удобным, как Perl, и более объектно-ориентированным, чем Python. Первая общедоступная версия 0.95 увидела свет в 1995 году. После этого Ruby быстро получил широкое распространение в Японии — это легко объяснимо происхождением языка и отсутствием языкового барьера между первыми пользователями Ruby и его создателем. За период с 95 по 2002 год в Японии вышло около двадцати книг о Ruby, и Ruby стал в этой стране более популярным языком, чем Python.

После 2000 года началось распространение Ruby по всему миру. Этому способствовало появление англоязычных книг, в первую очередь [1] и [3]. До 2004 года Ruby не был широко известен в Европе и США, и по популярности значительно уступал там Perl и Python. Однако благодаря своим качествам и большому количеству поддерживаемых платформ Ruby медленно, но верно умножал ряды своих приверженцев (среди которых в это время оказался и автор этих строк). Настоящий же всплеск интереса к Ruby спровоцировало появление Ruby-On-Rails (RoR) [5] — небольшого фреймворка для разработки Web-приложений. RoR стал для Ruby т.н. killer application, катализатором, благодаря которому сейчас Ruby получает признание во всем мире.

Влияние RoR на популярность Ruby неоднозначно. С одной стороны, благодаря RoR Ruby завоевывает причитающееся ему признание. Но с другой, складывается впечатление, что Ruby — это RoR, а RoR — это и есть Ruby. К счастью, это не так. Ruby — это динамически типизированный язык программирования, который начинал свою историю как скриптовый, но со временем превратился в более серьезный инструмент. Поэтому здесь рассказывается, в первую очередь, именно о языке Ruby, а RoR упоминается лишь по мере необходимости.

На момент написания этих строк язык Ruby находится на очень интересном этапе своего развития. До версии 1.8 он развивался, сохраненяя совместимость с предыдущими версиями. Но некоторое время назад разработчики Ruby, во главе с Якихиро Мацумото, решили, что для дальнейшего

движения вперед следует отказаться от 100% совместимости. Поэтому сейчас разработка Ruby разделилась на две ветви: поддержка стабильной версии 1.8.* (текущей версии Ruby) и создание новой версии 1.9.*, которая является предтечей следующей версии языка Ruby 2. Здесь описывается Ruby 1.8.*.

3 Начало работы с Ruby

Невозможно познакомиться с языком, не написав на нем ни одной строчки. И Ruby здесь не исключение. Поэтому в данном разделе приводится минимальная информация, необходимая для того, чтобы установить Ruby и начать эксперименты с ним.

По своей природе Ruby имеет очень низкий порог вхождения. Для начала работы достаточно только установленного интерпретатора Ruby. Простейшую программу, незабвенный "Hello, World", можно набрать и запустить непосредственно в интерпретаторе:

```
> ruby
puts "Hello, World\n"
^Z
Hello, World
```

или даже так:

```
> ruby -e'puts "Hello, world\n"'
Hello, world
```

Не нужно ни предварительной компиляции, ни линковки, что позволяет легко брать примеры из книг или документации, запускать их и экспериментировать с ними. И это один из лучших способов знакомства с языком. По крайней мере, для меня это оказалось именно так.

3.1 Где взять?

Исходные тексты и бинарные версии Ruby для ряда платформ доступны на официальном сайте языка Ruby [6]. На момент написания этих строк последней стабильной версией Ruby была версия 1.8.5.

Чтобы установить Ruby из исходных текстов под UNIX, достаточно распаковать загруженный архив ruby-1.8.5.tar.gz и выполнить обычную последовательность команд:

```
./configure
make
make install
```

В некоторых дистрибутивах Linux с развитой системой пакетов (например, Debian, Gentoo, SuSe, RedHat) Ruby доступен как уже подготовленный к инсталляции пакет, и для установки Ruby достаточно воспользоваться штатным механизмом инсталляции пакетов данного дистрибутива Linux.

Для Windows на сайте ruby-lang.org имеется предварительно скомпилированный вариант Ruby, инструкции по установке которого находятся в соответствующем файле README в архиве дистрибутива. Помимо этого для Windows имеется более простой и комфортный способ инсталляции Ruby — проект One-Click Installer [7]. Он удобен еще и тем, что, кроме самого интерпретатора Ruby и его стандартных библиотек, содержит еще и открытую IDE для Ruby (FreeRIDE [8]), набор дополнительных библиотек (в первую очередь RubyGems [9]) и электронный вариант первого издания книги "Programming Ruby".

Для проверки того, что Ruby установлен корректно, достаточно запустить интерпретатор ruby с ключом --version:

```
> ruby --version
ruby 1.8.5 (2006-08-25) [i386-mswin32]
```

Если вместо информации о версии и платформе будет выдано сообщение об ошибке, то, вероятно, нужно добавить путь к Ruby в переменную среды PATH.

Если Ruby устанавливается из стандартного дистрибутива, то очень вероятно, что RubyGems не входит в состав стандартной библиотеки (планируется сделать в одной из будущих версий Ruby).

ПРИМЕЧАНИЕ

RubyGems (англ. *gem* — драгоценный камень) — менеджер пакетов для языка программирования Руби, предоставляющий стандартный формат для программ и библиотек Руби (в самодостаточном формате «gems»), инструменты, предназначенные для простого управления установкой «gems», и сервер для их распространения (*из Википедии*).

Это не страшно, но лучше все-таки установить RubyGems, т.к. все больше и больше Ruby-библиотек и приложений распространяются в виде Gem-ов. Для этого достаточно загрузить дистрибутив RubyGems, распаковать его и выполнить в каталоге с распакованным дистрибутивом команду:

```
ruby setup.rb
```

после чего определить переменную среды RUBYOPT:

```
# Для Unix/bash.
export RUBYOPT="rubygems"

# Для Windows.
set RUBYOPT="rubygems"
```

Для работы с Ruby достаточно всего лишь приличного текстового редактора для программистов и интерпретатора Ruby. Но, если хочется работать в IDE, то можно обратить внимание на бесплатные FreeRIDE [8], Mondrian IDE [10] и RDT [11] (плагин к Eclipse), или платные Komodo [12] и Arachno [13]. В последнее время поддержка Ruby появляется и в других ориентированных на динамические языки IDE, поэтому запрос в Google по ключевым словам "Ruby IDE" даст гораздо более полный и актуальный список доступных Ruby IDE.

Отдельно следует упомянуть RubyForge.org [14] — аналог SourceForge [15] для Ruby-проектов. При необходимости найти какую-либо OpenSource-библиотеку для Ruby следует сначала обратиться к RubyForge.org. Кроме того, RubyForge.org по умолчанию является основным хранилищем RubyGem-ов и инсталляция подавляющего большинства оформленных в качестве Gem-ов Ruby-проектов осуществляется именно из этого хранилища.

3.2 Что запускать?

Ruby приложения выполняются с помощью интерпретатора, запускаемого командой "ruby":

```
  > ruby [<опции-ruby>] [имя-файла] [опции-программы]
```

Например:

```
  > ruby hello_world.rb
```

Интерпретатор ruby поддерживает набор опций, которые можно задать в командной строке. Их полный список можно получить, запустив ruby с ключом --help. На первом этапе наиболее важными из них могут оказаться следующие:

- -e ‘команда’, предписывает ruby выполнить указанную в параметре команду и завершить работу. Может использоваться для простых экспериментов, например, для проверки работы каких-либо методов. Самое простое применение — запуск ruby в качестве калькулятора:

```
> ruby -e’a=3; b=4; puts Math.sqrt(a*b)’
3.46410161513775
```

- -I<путь>, предписывает ruby искать подключаемые в программу модули в указанном каталоге. Пожалуй, самая нужна опция во время работы с ruby.
- -r<библиотека>, предписывает ruby загрузить указанную библиотеку до того, как будет загружена пользовательская программа. Очень важная, хотя и не часто использующаяся опция. С ее помощью, например, запускается штатный отладчик ruby:

```
> ruby -r debug hello_world.rb
```

и профайлер:

```
> ruby -r profile hello_world.rb
```

- -w, который включает режим выдачи предупреждений во время выполнения кода. С его помощью можно отлавливать потенциально опасные выражения и конструкции в Ruby-программах.

Если ruby запускается без имени файла, то ожидается, что код программы поступит из стандартного ввода. Это позволяет, например, запускать Ruby-программы, перенаправляя стандартный ввод:

```
> ruby < hello_world.rb
```

или с использованием какого-нибудь генератора программ (синтаксис, привычный для *nix):

```
> some_program_generator | ruby
```

или же вводить код непосредственно в интерпретаторе:

```
> ruby
include Math
a=3.0
b=4.0
c=sqrt(a*b)*sin(PI)
puts c
^Z
4.24216084818405e-016
```

Последний вариант, когда код вводится непосредственно в интерпретаторе, не очень удобен, т.к. в случае ошибки приходится набирать код заново. Однако в состав Ruby входит специальный инструмент, irb — Interactive Ruby, который делает интерактивное использование Ruby гораздо удобнее. Для работы с ним достаточно запустить команду irb, а затем вводить Ruby-инструкции. Irb будет выполнять их по мере ввода и показывать промежуточные результаты работы:

```
> irb
irb(main):001:0> include Math
=> Object
irb(main):002:0> a=3.0
=> 3.0
irb(main):003:0> b=4.0
=> 4.0
irb(main):004:0> c=sqrt(a*b)*sin(PI)
=> 4.24216084818405e-016
irb(main):005:0>
```

В реальной работе irb является незаменимым инструментом, возможно, даже более важным, чем отладчик. Irb позволяет очень легко экспериментировать с конструкциями языка, не прибегая к стандартной процедуре набора текста в редакторе и запуска программы в интерпретаторе. Зачастую часть кода сначала создается в сессиях irb, а уже затем переносится в программу. Например, очень полезен irb при работе с регулярными выражениями или сложными операциями над контейнерами.

3.3 Где искать информацию?

Информацию о самом языке лучше всего брать из англоязычных книг. По моему субъективному мнению, вне конкуренции здесь оба издания "Programming Ruby" ([1] и [2]). Из них предпочтительнее второе, в котором подробно описываются дополнительные инструменты (отладчик, irb, генератор документации rdoc, менеджер пакетов RubyGems), а также имеется отдельная часть, посвященная описанию деталей языка, очень важная для понимания того, как же работает вся магия Ruby. Справочником по языку выступает [16], которая свободно доступна в Internet. Можно отметить так же весьма оригинально написанную [4].

В качестве сборников практических советов по решению конкретных задач на Ruby следует обратить внимание на [3], [17] и [18].

Отдавая должное роли Ruby-On-Rails, следует упомянуть [19] и [20]. Тем более что первая из этих книг описывает самый удачный пока пример того, на что способно разумное использование возможностей языка Ruby. А вторая открывает секреты магии, благодаря которой это стало возможным.

Русскоязычную документацию по Ruby можно найти на [21]. Кроме самого описания языка, там присутствует раздел ссылок на другие русскоязычные источники.

Информацию об API стандартных (и не только) библиотек можно получать двумя основными способами. Во-первых, с помощью инструмента *ri — Ruby Information*. Он ищет описание указанного метода/класса в локально установленной документации:

```
> ri each_with_index
------------------------------------------- Enumerable#each_with_index
     enum.each_with_index {|obj, i| block }  -> enum
------------------------------------------------------------------------
     Calls _block_ with two arguments, the item and its index, for each
     item in _enum_.

        hash = Hash.new
        %w(cat dog wombat).each_with_index {|item, index|
          hash[item] = index
        }
        hash   #=> {"cat"=>0, "wombat"=>2, "dog"=>1}
```

Во-вторых, в Internet есть несколько ресурсов, на которых собрана документация по API многих Ruby-библиотек. В первую очередь это ruby-doc.org [22]. Кроме онлайн-версии документации, на ruby-doc.org есть возможность скачать ее в виде архива для офлайн-использования. Также ruby-doc.org позиционируется как базовая площадка для сбора Ruby-документации, поэтому на ruby-doc.org можно найти ссылки на различные Ruby-ресурсы и книги, посвященные Ruby.

Может быть полезен gemjack.com [23] на котором автоматически размещается информация о доступных на RubyForge.org Gem-ах.

Если нужной информации о чем-либо найти не удалось, то можно задать вопрос на ruby-talk [24].

4 Яркие грани

Данная глава посвящена краткому обзору некоторых граней языка Ruby. Они были выбраны на основании моих субъективных соображений, и перечисляются в почти случайном порядке, поскольку хотелось сделать на них акцент, а не написать краткий справочник по Ruby.

В этой главе приводится большое количество фрагментов Ruby-кода. Очень вероятно, что некоторые из них будут непонятны части читателей. Но я надеюсь, что степень непонимания будет снижаться по мере чтения. Также я рассчитываю, что большинство примеров не вызовет затруднений у читателей, имеющих опыт использования других динамических языков (т.к. Lua, Perl, Python, SmallTalk и др.).

Несколько слов о формате записи примеров. Комментарии в Ruby-программах начинаются с символа # и завершаются в конце строки:

```
# Это комментарий во всю строку.
a = 4 # А это комментарий в конце строки.
```

При приведении примеров Ruby-кода в документации принято использовать нотацию # => для указания результата выполнения выражения, как если бы после каждого выражения осуществлялась отладочная печать. Например, выполнение нескольких инструкций в irb приводит к результату:

```
irb(main):001:0> a = 4
=> 4
irb(main):002:0> if a > 0
irb(main):003:1> b = 'greater'
irb(main):004:1> else
irb(main):005:1* b = 'less or equal'
irb(main):006:1> end
=> "greater"
```

а запись того же самого результата непосредственно в Ruby-коде будет иметь вид:

```
a = 4                  # => 4
if a > 0
  b = 'greater'
else
  b = 'less or equal'
end                    # => 'greater'
```

4.1 Система именования

Система именований переменных, классов, методов, констант, атрибутов и пр., которая во многих языках присутствует как необязательная к исполнению рекомендация, зафиксирована в Ruby на уровне синтаксиса. Так, имя локальной переменной должно начинаться с маленькой буквы или подчеркивания, аналогично для имен методов. А вот константы и классы должны именоваться обязательно с заглавной буквы. Причем константы принято записывать полностью в верхнем регистре.

Имена глобальных переменных должны начинаться с символа $, имена нестатических атрибутов объекта (т.н. instance variables, переменных экземпляра) должны начинаться с @, а статических атрибутов (т.н. *class instance variables*) — с @@. Например:

```
$logger = nil                  # Имя глобальной переменной
class HelloPrinter             # Имя класса.
  HELLO_PREFIX = 'Hello, '     # Имя константы.

  @@all_names = Set.new        # Имя статического атрибута.

  def initialize(name)         # Имя метода и имя параметра.
    @name = name               # Имя атрибута.
    @@all_names << name
  end

  def print
```

```
      message = HELLO_PREFIX + @name    # Имя локальной переменной.
      $logger.log {
          'Printing hello for ' + @name + ', message: ' + message
      }
      puts message
    end
  end
```

На первый взгляд подобная система кажется неудобной и даже неэстетичной. Но это впечатление меняется на противоположное по мере привыкания к языку. Особенно удобно, на мой взгляд, использование префиксов @ для атрибутов объектов.

4.2 Наследие Perl

В начале своего развития Ruby многое позаимствовал из Perl. Сейчас некоторые из этих заимствований вызывают лишь досаду. Например, имена стандартных глобальных переменных Ruby: $0 содержит имя программы, заданное в командной строке интерпретатору, $: содержит список каталогов, в которых Ruby ищет подгружаемые модули, $_ содержит последнюю прочитанную с помощью gets строку и т.д.

Еще одно наследие Perl проявляется в неявном обновлении значений глобальных переменных при выполнении каких-либо операции. Например:

```
while gets
  if /Ruby/
    print
  end
end
```

является полностью корректной программой, принцип действия которой основан на том, что результат gets (чтения строки со стандартного ввода) сохраняется в $_. Затем значение $_ используется в поиске регулярного выражения /Ruby/ и, если поиск прошел успешно, $_ распечатывается методом print.

Подобное стремление к компактности Ruby-программ является пережитком прошлого (Perl-стиль), и сейчас подобные конструкции приводят к появлению предупреждений:

```
> ruby -w find_ruby_substring.rb danger_hello.rb
find_ruby_substring.rb:2: warning: regex literal in condition
```

и, возможно, будут изъяты из будущих версий языка. Сейчас сравнимая степень компактности достигается другими средствами, более выразительными и лучше воспринимаемыми. Этот же пример может быть переписан как:

```
print ARGF.grep(/Ruby/)
```

Неоднозначным, но до сих пор широко используемым наследием Perl является наличие постфиксных условных операторов и оператора unless:

```
f = some_data.find(pattern)    # find() возвращает nil, если
                               # pattern не найден.
unless f
  # Выполняется только если f == nil.
  ...
end

# постфиксный if, puts отрабатывает только если f != nil
puts "Found: #{f}" if f
# постфиксный unless, присваивание выполняется только если f == nil
f = SomeData.new unless f
```

Без таких операторов вполне можно было бы обойтись:

```
f = some_data.find(pattern)

if !f
  # Выполняется только если f == nil.
  ...
end

if f
  puts "Found: #{f}"
end

if !f
  f = SomeData.new
end
```

но в простых случаях отказ от использования постфиксных условных операторов увеличивает объем кода, поэтому часто постфиксные операторы оказываются очень удобными. К сожалению, при злоупотреблении ими программа быстро утрачивает читабельность, поэтому пользоваться постфиксными конструкциями нужно осторожно.

4.3 Структура программы и поиск модулей

Программа в Ruby — это текстовый файл, инструкции которого выполняются последовательно от начала до конца файла. Нет какой-либо специально выделенной точки входа в программу (функции main() как в C/C++ или статического метода main() как в Java). Например, в следующем случае:

```
a = 'Hello, '
b = 'World'

puts a
puts b

def concatenate(s1, s2)
  s1 + s2
end

puts concatenate(a, b)
```

Ruby сначала выполнит присваивание a и b, потом оттранслирует функцию concatenate и лишь затем выполнит ее вызов.

Этот обычный для интерпретаторов способ работы налагает на программиста некоторые ограничения. Например, при обработке исходного файла Ruby выполняет только элементарные проверки во время трансляции кода во внутреннее представление. Из-за этого перед исполнением программы может быть диагностирована только небольшая часть ошибок. Например, если функция concatenate содержит явную синтаксическую ошибку:

```
def concatenate(s1, s2)
  s1 +
end
```

то Ruby сможет ее диагностировать еще до выполнения первых инструкций:

```
> ruby prj_struct_sample_1-err.rb
prj_struct_sample_1-err.rb:9: syntax error
prj_struct_sample_1-err.rb:11: syntax error
```

Но если вместо явной синтаксической ошибки будет всего лишь опечатка в имени параметра:

```
def concatenate(s1, s2)
  s1 + s3
end
```

то такая ошибка будет обнаружена только во время первого вызова concatenate:

```
> ruby prj_struct_sample_1-err.rb
Hello,
World
prj_struct_sample_1-err.rb:8:in `concatenate': undefined local variable or
method `s3' for main:Object (NameError)
        from prj_struct_sample_1-err.rb:11
```

Более-менее крупные программы не следует размещать в одном большом исходном файле. Гораздо удобнее разделить код на несколько исходных файлов и подключать их по мере надобности. В Ruby подключение исходного файла выполняется с помощью метода require:

```
require 'fileutils'
```

Это аналог инструкции import из Java. Метод require ищет файл с расширением '.rb' и, если находит, то загружает его. Если .rb-файл не найден, то require пытается найти и загрузить динамически загружаемую библиотеку с учетом особенностей именования на целевой операционной системе (например, 'fileutils.dll' под Windows и 'libfileutils.so' под Unix). Если динамическая библиотека найдена, она загружается как расширение (предполагается, что она содержит код Ruby классов/методов, реализованных на C).

Метод require ищет файл в своих стандартных путях для поиска файлов, а если это не удалось — в текущем каталоге. Глобальная переменная $: содержит список имен каталогов, в которых Ruby будет осуществлять поиск подгружаемых модулей:

```
> ruby -e 'puts $:'
c:/ruby/lib/ruby/site_ruby/1.8
c:/ruby/lib/ruby/site_ruby/1.8/i386-msvcrt
c:/ruby/lib/ruby/site_ruby
c:/ruby/lib/ruby/1.8
c:/ruby/lib/ruby/1.8/i386-mswin32
.
```

Повлиять на этот список можно с помощью переменной среды RUBYLIB и аргумента командной строки '-I':

```
> set RUBYLIB=my-path
> ruby -I another-path -e 'puts $:'
another-path
my-path
c:/ruby/lib/ruby/site_ruby/1.8
c:/ruby/lib/ruby/site_ruby/1.8/i386-msvcrt
c:/ruby/lib/ruby/site_ruby
c:/ruby/lib/ruby/1.8
c:/ruby/lib/ruby/1.8/i386-mswin32
.
```

Управлять путями поиска модулей в Ruby приходится, если написанная на Ruby программа находится в каком-то нестандартном для Ruby каталоге. Например, пусть на Ruby написан какой-то инструмент и размещен в своем собственном каталоге, но вызывать его приходится из других каталогов. Тогда приходится писать либо вот так:

```
> ruby -I some/usefull/tool some/usefull/tool/main.rb
```

либо устанавливать переменную среды RUBYLIB:

```
> set RUBYLIB=some/usefull/tool
> ruby some/usefull/tool/main.rb
```

Но есть и более удобный способ решения вопроса о поиске файлов, загружаемых через метод require. Он основан на том, что Ruby использует глобальную переменную $: для перечисления имен каталогов, в которых будет осуществляться поиск. А раз это переменная, ее можно изменить:

```
> ruby -e '$:.unshift("yet-another-path"); puts $:'
yet-another-path
c:/ruby/lib/ruby/site_ruby/1.8
c:/ruby/lib/ruby/site_ruby/1.8/i386-msvcrt
c:/ruby/lib/ruby/site_ruby
c:/ruby/lib/ruby/1.8
c:/ruby/lib/ruby/1.8/i386-mswin32
.
```

т.е. процессом поиска частей программы можно управлять из самой Ruby-программы. Поэтому в Ruby-библиотеках распространен подход, когда в начале главного исходного файла (имя которого и передается интерпретатору в командной строке) содержится инструкция по изменению $: и несколько вызовов require, а весь остальной код находится в подключаемых файлах:

```
$:.unshift(File.dirname(__FILE__))

require 'module1'
require 'module2'
...
```

что позволяет вызывать Ruby-приложение без предварительной установки RUBYLIB или указания аргумента '-I':

```
> ruby some/usefull/tool/main.rb
```

Принцип работы конструкции

```
$:.unshift(File.dirname(__FILE__))
```

строится на том, что специальная константа __FILE__ содержит имя текущего исходного файла, а метод File::dirname выделяет из него имя каталога. В результате в $: помещается имя каталога, из которого было вызвано Ruby-приложение.

Кроме метода require в Ruby есть еще метод load. Различие между require и load состоит в следующем:

- require загружает модуль только однократно. Т.е. повторные вызовы require с тем же самым именем не приводят к повторной загрузке модуля. Метод load, напротив, загружает модуль столько раз, сколько раз load был вызван. В этом смысле load является аналогом #include из C/C++.
- require выполняет преобразование имени модуля (добавление расширений '.rb', '.dll' или '.so' по необходимости), а для load должно быть указано точное имя файла.

4.4 Строковые литералы и Symbol

В Ruby существует несколько способов записи строковых литералов. Сложно сказать, чем вызвано такое многообразие, к этой специфике нужно просто привыкнуть. Тем более что разные способы удобны для разных целей, и их разумное применение способно серьезно улучшить читабельность кода.

Первый из способов записи — это строки в одинарных кавычках:

```
a = 'Simple string'             # => Simple string
b = 'String with \\escapes'     # => String with \escapes
c = 'String with\nnewline?'     # => String with\nnewline?
```

В строках в одинарных кавычках обрабатываются всего две escape-последовательности: \\ заменяется на \, а \' на '. Поэтому самая распространенная сложность с такими строками — это указание в них последовательностей вроде \n, \t и др., которые не обрабатываются, а остаются в строке в неизменном виде.

Второй способ записи — это строки в двойных кавычках:

```
a = "Simple string"             # => Simple string
b = "String with \\escapes"     # => String with \escapes
c = "String with\nnewline?"     # => String with
                                # => newline?
```

В отличие от строк в одинарных кавычках, здесь обрабатываются все стандартные escape-последовательности. Но самым ценным в строках в двойных кавычках является возможность помещения в строку произвольного Ruby-выражения (т.н. string interpolation). Выражение заключается в специальные скобки #{ и }, и вычисляется во время работы со строкой:

```
a = 1                       # => 1
b = 2                       # => 2
c = "Sum is: #{a + b}"      # => "Sum is: 3"
```

В Ruby строковый литерал, записанный в одинарных или двойных кавычках, не может занимать несколько строк. Если нужно разбить длинный литерал на несколько строк, то следует использовать операцию конкатенации:

```
a = 'very long string in ' +
    'single quotes'             # => "very long string in single quotes"
b = "long string in " +
    "double " +
    "quotes"                    # => "long string in double quotes"
```

Кроме привычной записи строковых литералов в одинарных и двойных кавычках, в Ruby существует также возможность записи с помощью %-последовательностей. Строка в одинарных кавычках может быть записана через последовательность «%q», а в двойных кавычках — через «%Q» или просто %-последовательность. После префикса %q (%Q, %) должен следовать символ-терминатор, за которым располагается строковый литерал, ограниченный еще одним символом-терминатором. Если в качестве открывающего терминатора используются {, (или <, то закрывающим терминатором должен быть соответственно },) или >:

```
a = 1
b = 2
s1 = %q/Sum: #{a+b}, Sub: #{a-b}/  # => "Sum: \#{a+b}, Sub: \#{a-b}"
s2 = %q{Sum: #{a+b}, Sub: #{a-b}}  # => "Sum: \#{a+b}, Sub: \#{a-b}"
s2 = %q<Nested < ({ }) > >         # => "Nested < ({ }) > "

d1 = %Q/Sum: #{a+b}, Sub: #{a-b}/  # => "Sum: 3, Sub: -1"
d2 =  %/Sum: #{a+b}, Sub: #{a-b}/  # => "Sum: 3, Sub: -1"
d3 = %Q{Sum: #{a+b}, Sub: #{a-b}}  # => "Sum: 3, Sub: -1"
d4 =  %<Nested < ({ }) > >         # => "Nested < ({ }) > "
```

Существует еще один способ записи строковых литералов, называемый *here document*. Когда Ruby встречает в исходном тексте конструкцию <<ID или <<STR, он считает, что в последующих строках находится строковый литерал, ограниченный строкой, содержащей ID или STR:

```
banner = <<BANNER
Some useful tool.

Usage:
  tool [options] file
BANNER

help = <<'short help'
1. Introduction
2. Requirements
3. Usage
...
short help

banner  # => "Some useful tool.\n\nUsage:\n  tool [options] file\n"
help    # => "1. Introduction\n2. Requirements\n3. Usage\n...\n"
```

Как же используется это многообразие способов записи строковых литералов? Строки в одинарных кавычках удобно использовать для записи небольших строковых констант. Также, поскольку строки в одинарных кавычках не могут содержать Ruby-выражений, то во время исполнения программы строки в одинарных кавычках могут оказаться эффективнее, т.к. они не нуждаются в дополнительной обработке.

Строки в двойных кавычках используются практически повсеместно и, как правило, с использованием вложенных в них Ruby-выражений. Строки в двойных кавычках следует применять также, если в строку нужно поместить какую-то escape-последовательность.

Способы записи строковых литералов с помощью %-последовательностей и форм here document используются, когда в строку нужно преобразовать большой фрагмент текста, например, для сохранения в виде строки описания использования программы, или при подготовке фрагментов текста для последующего использования в функции eval:

```
def AbstractTarget.define_plural_form_method(singular_method)
  class_eval %Q{
    def #{singular_method}s(a)
      a.each { |e| #{singular_method}(e) }
    end
  }
end
```

Еще одной важной особенностью Ruby является наличие таких объектов, как *Symbol* (символ), записываемых в виде :ID или :STR. Символ — это специальный вариант строковой константы, для которой гарантируется, что все символы с одинаковым значением (в каком бы месте программы они не были записаны) будут представлены одним и тем же объектом. Например:

```
s1 = 'AAA'; s1.object_id        # => 20942000
s2 = "AAA"; s2.object_id        # => 20941490
s3 = "AAA"; s3.object_id        # => 20941300
s4 = :AAA ; s4.object_id        # => 190938
s5 = :'AAA'; s5.object_id       # => 190938
s6 = :A
s7 = "#{s6}#{s6}#{s6}".to_sym   # => :AAA
s7.object_id                    # => 190938
```

Здесь объекты s1, s2 и s3 являются разными объектами, что видно по их уникальным идентификаторам (идентификатор возвращается методом Object#object_id), несмотря на то, что содержат одинаковые значения. А вот ссылки s4, s5 и s7 ссылаются на один и тот же объект Symbol со значением :AAA, причем ссылка s7 образуется путем преобразования строки в Symbol посредством метода String#to_sym.

Символы очень широко используются в Ruby-программах. Ряд методов (например, attr_reader, attr_accessor, const_defined?, define_method и им подобных) принимают в качестве аргументов именно объекты Symbol, а не строки.

В отличие от строк, символы существуют в программе в единственном экземпляре и не убираются сборщиком мусора. Поэтому генерация слишком большого количества уникальных объектов Symbol в программе является своеобразной формой утечки памяти в Ruby.

4.5 Всё является объектом

Ruby является объектно-ориентированным языком и практически все в нем является объектом. Ruby-программы состоят из определения *классов*, создания *экземпляров* (*объектов*) и взаимодействия между объектами путем обмена сообщениями. Объект, которому отсылается сообщение, называется *получателем*. Получатель реализует обработку сообщения с помощью *метода*. Синтаксически запись отсылки сообщения выглядит как:

```
<получатель>.<сообщение>
```

например:

```
name.length
list.size
```

Такая запись в некоторых объектно-ориентированных языках рассматривается как вызов метода объекта. Этот термин можно использовать и в Ruby, но с одной важной оговоркой: в Ruby объекты могут обрабатывать даже те сообщения, имена которых не совпадают ни с одним из имен методов объекта. Подробнее эта возможность будет рассматриваться ниже, при обсуждении метода method_missing.

Правило «все является объектом» проявляется буквально на каждом шагу. Например, целочисленный литерал -1 является объектом, и у него можно вызывать методы:

```
-1.abs          # => 1
```

Вызов метода abs — это неявная посылка объекту сообщения abs. Ее можно сделать явной с помощью метода send, который есть у каждого объекта в Ruby:

```
-1.send(:abs)   # => 1
```

Определить класс, к которому принадлежит объект, можно с помощью метода class:

```
-1.class        # => Fixnum
```

Объектом может быть даже фрагмент кода:

```
hello = proc do puts "Hello, world" end
hello.class                                 # => Proc
hello.call                                  # => "Hello, world"
```

Здесь с помощью стандартного метода proc создается объект, содержащий код печати сообщения «Hello, world» (код обрамляется операторными скобками do/end или {}).

Объектная ориентированность Ruby идет еще дальше. Даже класс является объектом (как бы непривычно это не звучало в первый раз):

```
Fixnum.class      # => Class
```

Т.е. описание класса Fixnum является всего лишь объектом специального системного класса Class. В свою очередь класс Class является экземпляром класса Class, т.е. самого себя. Осознать и привыкнуть к этому не так-то просто, но, к счастью, добираться до таких глубин Ruby приходится крайне редко. Очень подробно детали взаимоотношений объект/класс описаны в [2].

То, что класс является объектом, проявляется в способе создания экземпляров класса. Для этого классу отсылается сообщение (вызывается метод) new, который и конструирует новый объект:

```
s = String.new('AAA')
```

Здесь объектом-получателем сообщения является объект (он же класс) String. А результатом обработки сообщения new является новый объект класса String.

4.6 Метод inspect у каждого объекта

У каждого объекта Ruby можно вызвать метод *inspect*, возвращаемым значением которого является строка с информацией о данном объекте:

```
1.inspect                          # => "1"
(0.3).inspect                      # => "0.3"
:Hello.inspect                     # => ":Hello"
[ 1, 2, 3 ].inspect                # => "[1, 2, 3]"
{ :first => 1 }.inspect            # => "{:first=>1}"

File.open('_vimrc', 'r').inspect   # => "#<File:_vimrc>"
```

Метод inspect очень удобен при организации отладочных печатей в программе. Его использование в сочетании с отображением возвращаемого значения на стандартный поток вывода является настолько общеупотребительным, что в Ruby есть специальный метод *Kernel#p*:

```
array = [ 1, 2, 3 ]
p array
```

что является сокращенной формой записи:

```
puts array.inspect
```

Небольшой иллюстрацией востребованности метода inspect является то, что подавляющее большинство значений, показанных в приведенных здесь примерах после символов #=>, получено как раз с помощью метода inspect.

4.7 Все выражения имеют значения

Выражения (такие как if, case, циклы и пр.) имеют значения. Например, можно написать:

```
x = if condition-1
      value-1
    elsif condition-2
      value-2
    else
      value-3
    end
```

Переменная x получит значение, зависящее от того, по какой ветке if пойдет вычисление.

В случае циклов while и loop возвращаемым значением будет nil. Т.е. смысла в получении возвращаемого значения такого цикла нет, но синтаксис языка все равно позволяет записать и успешно выполнить, например, такую программу:

```
i = 0;
x = while i < 10
      puts '*' * i
      i += 1
    end
x                                       # => nil
```

В случае же цикла for возвращаемым значением будет объект-условие:

```
x = for i in 1..5
      puts '*' * i
    end
x                                       # => 1..5
```

Значение nil возвращают также такие выражения, как объявления методов и даже классов:

```
c = class My
      def hello; puts 'Hello!'; end
    end
c                                       # => nil

m = def hello
      puts 'Hello!'
    end
m                                       # => nil
```

Возвращаемые значения nil для циклов и деклараций классов/методов выглядят экзотикой, однако Ruby — динамический язык, и он позволяет вычислять произвольные выражения с помощью функции eval:

```
str = 'a = 1; b = 3; a + b'
x = eval(str)
x                                       # => 4
```

Поскольку eval может получить строку, в которой содержится цикл или декларация класса, то и в этом случае eval должен возвратить корректное и осмысленное значение. Таким значением как раз и является nil.

То, что любое выражение в Ruby имеет значение, делает использование ключевого слова return очень редким делом. Функция (метод) в Ruby возвращает значение последнего выполненного в ней выражения. Поэтому, например, простейший метод-getter в Ruby выглядит так:

```
def get_something
  something
end
```

Использовать return приходится, только если работу функции требуется прервать раньше времени. Например:

```
def some_calculation
  # Возвращаем уже вычисленное выражение, если оно закэшированно.
  return @value if @value

  # Значение еще не вычислялось, делаем это.
  ...
```

```
  @value = ... # Результат этого присваивания одновременно будет
               # возвращаемым значением функции.
end
```

4.8 Классы

Классы в Ruby похожи на классы в других объектно-ориентированных языках, но есть некоторые интересные особенности. Например, пусть следующий класс описывает параметры подключения к удаленному хосту:

```
class ConnectionParams
  attr_accessor :host
  attr_accessor :user
  attr_accessor :password

  def initialize(host, user, password)
    @host = host
    @user = user
    @password = password
  end
end
```

Такое небольшое объявление создает класс ConnectionParams, объекты которого будут иметь три атрибута (имя хоста, имя и пароль пользователя), конструктор с тремя параметрами и методы получения/изменения хранящихся в объекте значений:

```
params = ConnectionParams.new('localhost', 'hacker', 'secret')
params.inspect                          # => "#<ConnectionParams:0x27f2090
                                              @password=\"secret\",
                                              @user=\"hacker\",
                                              @host=\"localhost\">"

params.host                             # => "localhost"

params.host = 'localhost:3333'          # => "localhost:3333"

params.inspect                          # => "#<ConnectionParams:0x27f2090
                                              @password=\"secret\",
                                              @user=\"hacker\",
                                              @host=\"localhost:3333\">"
```

Конструктор класса в Ruby — это метод со специальным именем *initialize*, который автоматически вызывается для инициализации нового экземпляра после того, как интерпретатор Ruby выделит экземпляру память. Конструирование нового экземпляра и вызов его конструктора в Ruby объединяется в одну операцию — вызов метода *new* у класса создаваемого объекта. В данном случае это метод new с тремя параметрами для класса ConnectionParams.

Класс ConnectionParams имеет три атрибута: @host, @user и @password. Атрибутами объекта их делает наличие префикса @ — любое имя с таким префиксом становится атрибутом того объекта, в методе которого имя встретилось впервые. Соответственно, атрибут создается при первом упоминании его имени. Что иногда становится причиной ошибок. Например, в ConnectionParams добавляется метод:

```
def switch_user(user, password)
  @usr = user
  @password = password
end
```

в котором допущена элементарная описка — вместо имени @user набрано @usr. Интерпретатор Ruby, встретив новое имя атрибута @usr, создаст новый атрибут и инициализирует его новым значением:

```
params = ConnectionParams.new('localhost', 'hacker', 'secret')
params.inspect                    # => "#<ConnectionParams:0x27f28b0
                                        @password=\"secret\",
                                        @user=\"hacker\",
                                        @host=\"localhost\">"
params.switch_user('hacker2', 'topsecret')
params.inspect                    # => "#<ConnectionParams:0x27f28b0
                                        @password=\"topsecret\",
                                        @user=\"hacker\",
                                        @usr=\"hacker2\",
                                        @host=\"localhost\">"
```

Такое поведение, несмотря на периодическое возникновение подобных ошибок, является следствием динамической природы языка Ruby. Это его особенность (существующая и в некоторых других динамических языках), с ней нужно просто считаться. Тем более что в некоторых случаях ее можно успешно использовать.

Формально говоря, члены класса, имена которых начинаются с префикса @, называются переменными экземпляра (instance variables) — т.е. это переменные, которые присутствуют в каждом экземпляре (объекте) класса. Атрибутами же называются те переменные экземпляров, которые доступны пользователю класса посредством методов получения и изменения значения переменной экземпляра. Так, объявление:

```
class MyClass
  def initialize
    @attr = 'value'
  end
end
```

создает переменную экземпляра @attr, доступа к которой пользователи класса MyClass не имеют. Чтобы исправить это, нужно создать метод getter, который будет возвращать значение @attr:

```
def attr
  @attr
end
```

и метод setter, который будет присваивать @attr новое значение:

```
def attr=(v)
  @attr = v
end
```

что позволяет создать впечатление, что attr — это общедоступный атрибут класса MyClass:

```
m = MyClass.new
m.attr                # => "value"
m.attr = 'new value'
m.attr                # => "new value"
```

Такое именование getter/setter-ов является стандартным соглашением в Ruby, напоминающим свойства (properties) в некоторых языках программирования. И, поскольку писать подобные getter/setter-ы приходится довольно часто, в Ruby добавлен специальный синтаксический сахар, продемонстрированный выше в классе ConnectionParams: конструкция *attr_accessor* автоматически добавляет в класс реализацию getter/setter-а для указанного атрибута. Также существуют аналогичные *attr_reader* (объявляет только getter) и *attr_writter* (объявляет только setter).

Конструктор класса, getter/setter-ы и другие подобные методы по аналогии с переменными экземпляра (instance variables) называются методами экземпляров (instance methods), поскольку они вызываются для экземпляров (объектов) класса. Но в Ruby есть еще две категории переменных и методов для классов, которые эквивалентны понятию статических атрибутов/методов в других

языках программирования — переменные класса и методы класса (*class instance variables* и *class instance methods*):

```
class ConnectionParams
  ...
  # Номер TCP/IP порта по умолчанию.
  @@default_port = 3333

  # Проверка наличия номера порта в строке host и
  # автоматическое добавление номера порта в строку в
  # случае отсутствия.
  def ConnectionParams.ensure_port_specified(host)
    ...
  end
  ...
end
```

Имя переменной класса начинается с префикса @@. В отличие от переменной экземпляра переменная класса должна быть обязательно объявлена и проинициализирована перед использованием. При определении метода класса сначала указывается имя класса, и только затем имя метода — таков синтаксис для методов класса в Ruby (строго говоря, в Ruby есть еще несколько способов определения метода класса, но это, вероятно, наиболее простой из них).

Так же, как статические атрибуты/методы в других языках, переменные класса разделяются всеми экземплярами класса. Но вот в чем проявляется особенность Ruby, так это в контексте, в котором работает каждый из вышеуказанных видов методов. Ruby — это чистый объектно-ориентированный язык, в нем есть только методы объектов, и каждый метод вызывается в контексте какого-то объекта. Соответственно, в каждом методе есть неявная ссылка *self*, связывающая метод с его объектом. И самая интересная часть отличий между методами экземпляра и методами класса заключается как раз в значении ссылки self:

```
class MyClass
  def instance_method
    self.class.name
  end

  def MyClass.class_method
    self.class.name
  end
end

m = MyClass.new
m.instance_method               # => "MyClass"
MyClass.class_method            # => "Class"
```

Метод class для объекта в Ruby возвращает ссылку на описание класса, которому принадлежит объект. Соответственно, конструкция self.class.name определяет имя класса, которому принадлежит объект self. Вполне ожидаемо, что внутри instance_method self — это объект класса MyClass. А вот внутри class_method self является экземпляром объекта класса Class. Что же это за объект?

Чтобы ответить на этот вопрос, нужно еще раз вспомнить, что Ruby — это чистый объектно-ориентированный язык, в котором *все* является объектом. В том числе и описания классов. Поэтому описание класса MyClass в Ruby также представляется объектом (экземпляром специального класса Class). Этот объект создается в момент объявления:

```
class MyClass
```

и он же является значением self в методах класса.

Из вышесказанного следует, что методы экземпляра работают в контексте конкретного экземпляра (т.к. для них self указывает на экземпляр), а методы класса в контексте класса (т.к. для них self

указывает на класс). Но в Ruby интересно еще и то, что контекст класса как бы «открывается» ключевым словом class и «закрывается» ключевым словом end в описании класса. Т.е. между class и end также есть self, и он имеет то же самое значение, что внутри метода класса:

```
class MyClass
  def MyClass.class_method
    self.object_id
  end

  self.object_id                  # => 20947690
end

MyClass.class_method              # => 20947690
```

Следующей интересной особенностью Ruby является то, что внутри описания класса можно использовать не только декларации методов/констант/переменных, но и любые другие выражения: if, while, for, case и т.д. Ruby подходит к определению класса так же, как к интерпретации любой другой части программы — Ruby просто выполняет встречаемые инструкции. Соответственно, внутри описания класса Ruby позволяет вызывать методы класса (именно методы класса, а не методы экземпляра, поскольку контекстом является класс):

```
class MyClass
  def MyClass.class_method
    self.object_id
  end

  class_method                    # => 20947690
end
```

Здесь становится видна одна из ролей методов класса — поскольку они запускаются в контексте класса, они могут использоваться для воздействия на класс. Примером чего являются уже упоминавшиеся выше конструкции attr_accessor/attr_reader/attr_writer — это обычные методы класса, которые расширяют описание того класса, в котором были использованы, посредством добавления методов getter-/setter-ов.

4.9 Наследование

Ruby поддерживает только одиночное наследование, и все классы образуют единую иерархию с одним общим корнем — классом *Object*.

Базовый класс указывается при помощи специального синтаксиса во время описания класса:

```
class Derived < Base
  ...
end
```

В этом примере класс Derived произведен от класса Base.

Определить суперкласс любого класса можно с помощью метода superclass:

```
class Base
end

class Derived < Base
end

Base.superclass                   # => Object
Derived.superclass                # => Base
```

Этот пример показывает, что если во время описания явно не задать имя суперкласса (как в случае класса Base), то Ruby автоматически использует в качестве суперкласса Object.

Как и в других объектно-ориентированных языках, наследование в Ruby используется для того, чтобы производные классы расширяли или изменяли поведение своих суперклассов. Это достигается за счет перекрытия методов базового класса и внесения в производный класс собственных методов. В этом смысле Ruby не преподносит разработчику каких-либо сюрпризов. Но некоторые особенности, связанные с наследованием, все-таки есть.

Например, если производный класс определяет собственный конструктор, то конструктор базового типа автоматически не вызывается:

```
class Base
  def initialize
    puts "Base#initialize"
  end
end

class Derived < Base
  def initialize
    puts "Derived#initialize"
  end
end

Derived.new
```

Этот код приводит к печати сообщения:

```
Derived#initialize
```

Т.е. конструктор базового класса управление не получил, поскольку это нужно делать явно с помощью ключевого слова super:

```
class Derived < Base
  def initialize
    puts "Derived#initialize"
    super
  end
end
```

Что приводит к вызову конструктора базового класса:

```
Derived#initialize
Base#initialize
```

За вызов конструктора базового класса отвечает производный класс, и этот вызов может быть выполнен в любой момент. В сочетании с еще одной особенностью наследования это может приводить к интересным эффектам.

Дело в том, что в Ruby все методы класса являются, в терминологии других языков, виртуальными. Т.е. любой метод можно переопределить в производном классе:

```
class Base
  def first
    "Base#first"
  end

  def second
    "Base#second"
  end
```

```
end

class Derived < Base
  def first
    "Derived#first"
  end
end

d = Derived.new
d.first                          # => "Derived#first"
d.second                         # => "Base#second"

class AnotherDerived < Derived
  def second
    "AnotherDerived#second"
  end
end

a = AnotherDerived.new
a.first                          # => "Derived#first"
a.second                         # => "AnotherDerived#second"
```

Здесь видно, как класс Derived переопределил метод first, но унаследовал без изменений метод second. В свою очередь, класс AnotherDerived переопределил метод second, но оставил без изменения метод first.

Ситуация усугубляется еще и тем, что в Ruby нет возможности указать, что какой-то конкретный метод больше не может быть переопределен (т.е. в Ruby нет аналога ключевого слова final из Java).

Когда для объекта какого-то класса вызывается метод или, говоря более точно, объекту отсылается сообщение, Ruby ищет обработчик этого сообщения (т.е. тело метода) по имени сообщения. Поиск начинается с того класса, которому принадлежит объект. Если в этом классе соответствующий обработчик не найден, то поиск продолжается в непосредственном суперклассе, затем в суперклассе суперкласса и т.д. Неважно, в каком контексте производится вызов метода — происходит ли это в каком-то методе самого объекта, в каком-то методе базового класса или где-то еще. Ruby всегда точно знает класс объекта, и поиск метода начинается именно в этом классе.

Такое поведение приводит к тому, что если в конструкторе суперкласса вызывается метод, перекрытый в производном классе, то вызывается именно перекрытая производным классом версия метода:

```
class Base
  def initialize
    puts self.class.name
    puts first
    puts second
  end

  def first
    "Base#first"
  end

  def second
    "Base#second"
  end
end

class Derived < Base
  def first
    "Derived#first"
  end
end

d = Derived.new
```

что приводит к:

```
Derived
Derived#first
Base#second
```

т.е. сначала Base#initialize печатает имя класса, к которому реально принадлежит объект (Derived), затем результаты методов Derived#first и Base#second.

Такое поведение в корне отличается от подхода, принятого, например, в C++. В C++ в конструкторе базового класса будут вызываться только те версии виртуальных методов, которые определены именно в этом базовом классе. В C++ это объясняется тем, что конструкторы вызываются в строгой последовательности от базового к производному, и в момент работы конструктора базового класса атрибуты производного класса еще не инициализированы. Поэтому вызов виртуального метода из производного класса может привести к работе с неинициализированными атрибутами.

В Ruby такого контроля нет. Поэтому разработчик может столкнуться с тем, что его код работает не так, как предполагалось:

```
class ConnectionParams
  attr_reader :host, :user, :password

  def initialize(host, user, password)
    @host, @user, @password = host, user, password

    # Метод setup_default назначит значения по умолчанию тем атрибутам,
    # которые получили значение nil от пользователя.
    setup_defaults
  end

  def setup_defaults
    # Если не задан @host, то используется localhost.
    @host = 'localhost' unless @host
  end
end

class SSLConnectionParams < ConnectionParams
  attr_reader :cert, :ca_cert

  def initialize(
      host, user, password,
      cert, ca_cert,
      default_cert = '~/user.pem',
      default_ca_cert = '~/ca.pem')
    # Внимание: в конструкторе базового класса произойдет
    # вызов setup_defaults.
    super(host, user, password)

    @cert, @ca_cert = cert, ca_cert
    @default_cert, @default_ca_cert = default_cert, default_ca_cert
  end

  def setup_defaults
    # Выполнение действий базового класса...
    super

    # ...а затем собственных.
    @cert = @default_cert unless @cert
    @ca_cert = @default_ca_cert unless @default_ca_cert
  end
end
```

```
ssl = SSLConnectionParams.new(nil, 'user', 'pwd', nil, nil)
ssl.host                          # => "localhost"
ssl.cert                          # => nil
ssl.ca_cert                       # => nil
```

В данном случае делалась попытка позволить пользователю не задавать явно параметр host в конструкторе ConnectionParams, и параметры cert, ca_cert в конструкторе SSLConnectionParams. Предполагалось, что конструктор посредством метода setup_defaults при необходимости подставит значения по умолчанию.

Для атрибута ConnectionParams#host это сработало, но в случае с SSLConnectionParams#cert/ca_cert произошла ошибка, от которой разработчик в C++ защищен — переопределенный метод SSLConnectionParams#setup_defaults был вызван из конструктора базового класса еще до того, как конструктор SSLConnectionParams проинициализировал атрибуты класса SSLConnectionParams.

В методе SSLConnectionParams#setup_default не произошло никакой ошибки при обращении к неизвестным на тот момент атрибутам cert/default_cert и ca_cert/default_ca_cert из-за того, что Ruby определяет атрибуты в объекте при первом обращении к ним. При этом, если атрибут стоит в правой части присваивания (как в случае с default_cert/default_ca_cert), то в качестве значения атрибута берется nil.

Тем не менее, несмотря на потенциальные опасности, ручной вызов конструктора базового класса в произвольный момент и вызов в базовом классе самых последних версий переопределенных методов имеют и положительные стороны. Например, пусть есть иерархия классов для отображения элементов текстового документа на экране. У каждого элемента есть свой набор атрибутов, таких как гарнитура шрифта, размер, вид (полужирный, курсив и т.д.). Все эти параметры будут присутствовать у всех элементов, поэтому можно определить базовый класс, например, ElementPainter, который в своем конструкторе выполнит создание необходимых графических ресурсов (или породит исключение, если это не удалось). А производные классы будут сообщать ему свои специфические настройки посредством переопределенных методов. Например:

```
class ElementPainter
  def initialize
    # Конструирование графических объектов с помощью методов
    # font_family, font_size, bold?, italic?, underlined?
    ...
  end

  # Производные классы будут переопределять эти методы.
  def font_family; 'Times New Roman'; end
  def font_size; 10; end
  def bold?; false; end
  def italic?; false; end
  def underlined? false; end
  ...
end

# Заголовки частей, глав, разделов.
# Все они рисуются полужирным шрифтом, но разных размеров.
class HeaderElementPainter < ElementPainter
  def bold?; true; end
end

class PartHeaderElementPainter < HeaderElementPainter
  def font_size; 16; end
end

class ChapterHeaderElementPainter < HeaderElementPainter
  def font_size; 14; end
end

class SectionHeaderElementPainter < HeaderElementPainter
```

```
  def fond_size; 12; end
end

# Примеры кода рисуются шрифтом Courier New размера 8 пунктов.
class SourceCodeElementPainter < ElementPainter
  def font_family; 'Courier New'; end
  def font_size; 8; end
end
...
```

При этом во время создания объекта, производного от класса ElementPainter, все операции по созданию графических ресурсов выполняет сам класс ElementPainter, но использует информацию из своих производных классов.

4.10 Модули и Mixin-ы

Еще одним краеугольным камнем Ruby, тесно связанным с классами и наследованием, является механизм модулей Ruby. Модули играют в языке двойную роль. Во-первых, модули образуют самостоятельные пространства имен (по аналогии с пространствами имен в C++):

```
module Statistics
  def Statistics.print
    ...
  end
end

module UsageHelp
  def UsageHelp.print
    ...
  end
end

Statistics::print
UsageHelp::print
```

В роли пространств имен модули Ruby позволяют давать описанным в модулях константам, классам и методам короткие имена, возможно, пересекающиеся с такими же именами в других модулях. Но такое пересечение не вызывает проблем, поскольку для доступа к имени нужно указывать имя модуля (которое можно опускать, если вызов делается в рамках того же модуля).

В приведенном выше примере показано объявление методов класса (т.н. class instance methods) в модулях. Это обычная практика, когда модули создаются для того, чтобы играть роль пространств имен. Но в модулях можно объявлять и методы экземпляра (т.н. instance methods) в которых можно обращаться к переменным экземпляра (т.н. instance variables):

```
module Statistics
  attr_reader :hits_count

  def increment_hits_count
    @hits_count ||= 0
    @hits_count += 1
  end
end
```

Такие методы модуля станут доступными для использования только после того, как модуль Statistics будет *подмешан* в какой-либо класс (т.е. сыграет роль *примеси* (mixin)):

```
class Cache
  include Statistics
```

```
  def get_page(page_info)
    if page_in_cache?(page_info)
      increment_hits_count
      ...
    else
      ...
    end
  end
  ...
end

c = Cache.new
c.hits_count
```

Конструкция:

```
include Statistics
```

является инструкцией по подмешиванию содержимого модуля в класс. В результате подмешивания все методы экземпляра из модуля становятся методами экземпляра того класса, который выполнил подмешивание. Соответственно, методы класса из подмешиваемого модуля становятся методами класса.

После того, как модуль подмешан в класс, он начинает играть роль еще одного суперкласса. В первую очередь это сказывается на поиске методов объекта: сначала производится поиск методов в том классе, которому принадлежит объект, затем во всех подмешанных в класс модулях, затем в суперклассе и т.д. При этом «официальным» суперклассом для класса продолжает оставаться тот класс, который был указан при описании класса (его имя возвращается методом superclass), зато в списке примесей появляются все подмешанные в класс модули (этот список можно получить методом ancestors):

```
puts c.class.superclass, '-'*10, c.class.ancestors
```

что для объекта Cache дает:

```
Object
----------
Cache
Statistics
Object
Kernel
```

т.е. суперклассом продолжает оставаться класс Object, а вот в списке модулей виден модуль Statistics (метод ancestors включает в список и имена тех модулей/классов, для которых он вызывается, поэтому в списке присутствует имя Cache; имя Kernel находится в списке потому, что модуль Kernel подмешивается к классу Object). Причем возвращенный методом ancestors список как раз показывает порядок, в котором Ruby будет искать методы объекта класса Cache.

Поскольку при подмешивании модуль как бы становится суперклассом, в подмешавшем его классе появляется возможность переопределить любой из методов модуля. Например, класс Cache может предоставить свою реализацию метода increment_hits_count:

```
def increment_hits_count
  super
  if 0 == hits_count % 100
    store_cache
  end
end
```

при этом обращение к super будет передавать управление реализации increment_hits_count в модуле Statistics.

Ситуация с переменными экземпляра (т.н. instance variables), которые вводятся в модуле, похожа на ситуацию с методами экземпляра. Но здесь есть несколько важных моментов.

Во-первых, в модуле нет конструктора, в котором переменные экземпляра можно было бы должным образом проинициализировать. Именно поэтому первая строка приведенного выше метода Statistis#increment_hits_count содержит инициализацию @hits_count на случай, если эта инициализация еще не была выполнена (в противном случае попытка добавления к @hits_count единицы приведет к ошибке, т.к. для объекта nil нет метода сложения с целым числом).

Во-вторых, нет средств контроля за тем, чтобы имена переменных экземпляра в модуле не пересекались с именами переменных экземпляра в подмешивающем модуль классе. Например, если модуль Statistics и класс Cache разрабатывали разные программисты, каждый из них мог определить у себя переменную экземпляра с именем @hits_count, не зная, что это имя уже занято. При подмешивании модуля Statistics в класс Cache никаких ошибок или предупреждений программисту выдано не будет, что в результате может привести к трудноуловимым ошибкам. Однако эта ситуация не является особенностью именно модулей — аналогичные проблемы можно получить и при наследовании, если производный класс начнет по-своему использовать имя переменной экземпляра из какого-то базового класса.

Использование модулей в качестве примесей является широко распространенной практикой в Ruby. Эта практика прослеживается даже в стандартной библиотеке. Хрестоматийными примерами использования примесей являются модули Comparable [35] и Enumerable [36]. Первый позволяет в результате подмешивания определить в классе набор методов для сравнения объекта (<, >, <=, =>, ==, between?). Для этого в подмешивающем классе достаточно определить метод <=>. Например, следующий класс, описывающий размер комнаты (длину, ширину и высоту), посредством подмешивания модуля Comparable позволяет сравнивать комнаты между собой по объему:

```
class Room
  include Comparable

  attr_reader :width, :length, :height

  def initialize(width, length, height)
    @width, @length, @height = width, length, height
  end

  # Объем комнаты.
  def bulk
    @width * @length * @height
  end

  # Должен возвращать -1, если self < other;
  # 0, если self == other;
  # 1, если self > other.
  def <=>(other)
    b1, b2 = bulk, other.bulk
    if b1 < b2 then -1; elsif b1 > b2 then 1 else 0 end
  end

  def to_s
    "Room: #{@width}x#{@length}x#{height}"
  end
end

small = Room.new(3, 3, 2.05)
big = Room.new(4, 10, 3)

big < small                    # => false
small < big                    # => true
```

Модуль Enumerable позволяет расширить класс методами, присущими коллекциям (например, методами max, min, find, find_all, map и пр.). Для этого подмешивающий класс должен предоставить метод each, который последовательно перебирает все элементы коллекции. Например, следующий класс Floor (этаж) содержит список комнат (экземпляров Room) и посредством подмешивания Enumerable позволяет работать с этим списком как будто класс Floor является коллекций объектов Room:

```
class Floor
  include Enumerable

  def initialize
    @rooms = []
  end

  def add(room)
    @rooms << room
  end

  def each(&block)
    @rooms.each(&block)
  end
end

floor = Floor.new
floor.add(Room.new(5, 10, 2.05))
floor.add(Room.new(4, 10, 2.05))
floor.add(Room.new(6, 10, 2.05))

# Поиск самой маленькой комнаты.
puts "min: #{floor.min}"
# Поиск самой большой комнаты.
puts "max: #{floor.max}"
# Поиск все комнат, чья ширина меньше 6 метров.
puts "width < 6: #{floor.find_all do |r| r.width < 6 end.join('; ')}"
```

что приводит к:

```
min: Room: 4x10x2.05
max: Room: 6x10x2.05
width < 6: Room: 5x10x2.05; Room: 4x10x2.05
```

4.11 Еще раз: все является объектом

То, что в Ruby все является объектом, приводит к еще одной интересной особенности. Описание класса в Ruby — это объект специального класса Class, который содержит список переменных и методов экземпляра, а также переменные и методы класса. Доступ к этому объекту в Ruby осуществляется через имя класса. Но особенность Ruby в том, что имя класса — это всего лишь имя константы, значением которой является объект-описатель класса. Т.е. конструкция:

```
class My
  ...
end
```

приводит к определению двух сущностей: объекта-описателя класса «My» (который существует где-то в интерпретаторе Ruby) и константы My, ссылающейся на объект-описатель.

То, что класс является объектом, позволяет делать в программе интересные вещи. Например, классы можно присваивать переменным:

```
class PrettyPrinter; ... end
class DraftPrinter; ... end

printer = use_pretty_print? ? PrettyPrinter : DraftPrinter
...
p = printer.new(...)
```

или реализовать фабрику объектов, которая будет получать класс конструируемых объектов в качестве параметра:

```
def factory(klass, *args)
  klass.new(*args)
end

s = factory(String, 'my string')
s.inspect                          # => "\"my string\""

a = factory(Array, 2, s)
a.inspect                          # => "[\"my string\", \"my string\"]"
```

или даже возвращать классы из методов как возвращаемые значения:

```
def detect_printer(options)
  options.use_pretty_print? ? PrettyPrinter : DraftPrinter
end
...
class CustomPrinter < detect_printer(options)
  def print; ... end
end
```

4.12 Базовые типы Array и Hash

Прежде, чем переходить к обсуждению следующей грани языка Ruby, необходимо остановиться на двух чрезвычайно важных базовых типах: *Array* и *Hash*.

Оба класса являются упорядоченными коллекциями объектов. Только в Array ключом для доступа к объекту является порядковый номер (индекс) элемента. А в Hash элементы хранятся в виде пар объектов <ключ, значение>, где для доступа к значению требуется использование объекта-ключа.

Распространенным способом записи значений Array и Hash является использование специальных литералов. Так, значение Array записывается в виде последовательности элементов, заключенных в квадратные скобки:

```
e = []                    # Пустой Array.
a = [ 1, 2, 3 ]           # Array с тремя значениями.
w = [ 'one', 'two' ]      # Array с двумя значениями.
m = [ 1, 'First', Time.now, w ]
```

Значение Hash записывается в виде последовательности элементов, заключенных в фигурные скобки. При этом количество элементов должно быть четным. Элементы должны быть разделены запятыми или, что более предпочтительно, каждая пара элементов записывается с помощью специальной нотации =>:

```
e = {}                          # Пустой Hash.
d = { 1, 'one', 2, 'two' }      # Hash с двумя парами значений, где ключами
                                # являются целые числа.
t = { :today     => Time.now,
      :yesterday => Time.now - 86400,
      :tomorrow  => Time.now + 86400 } # В качестве ключей используются
                                       # объекты Symbol.
```

Доступ к хранящимся в коллекции элементам осуществляется с помощью метода [], для которого в случае Array аргументом является индекс элемента (индексация элементов Array в Ruby начинается с 0), а в случае Hash — объект-ключ:

```
a[ 0 ]                          # => 1
w[ 1 ]                          # => 'two'
m[ 3 ]                          # => [ 'one', 'two' ]
d[ 1 ]                          # => 'one'
t[ :tomorrow ]                  # => Wed Jan 03 10:01:46 +0300 2007
```

При доступе к несуществующему элементу в Hash по умолчанию возвращается значение nil:

```
t[ :DayAfterTomorrow ]          # => nil
```

но его можно изменить, если задать значение по умолчанию в конструкторе Hash:

```
t = Hash.new(0)
t[ :DayAfterTomorrow ]          # => 0
```

Объекты типа Array и Hash могут содержать в себе объекты разных типов — это следствие динамической природы языка. Более того, объект Hash может использовать ключи разных типов. Это возможно, поскольку Hash является реализацией контейнера «хэш-таблица» и объекты-ключи должны всего лишь предоставлять метод *hash* для вычисления хэш-кода объекта и метод *eql?* для проверки равенства двух объектов. До тех пор, пока объекты разных типов производят разные хэш-коды и корректно реализуют eql?, они могут служить ключами для Hash. Главным ограничением здесь является то, что Hash не может содержать двух одинаковых ключей.

Классы Array и Hash содержат большое количество методов, часть из которых наследуется из mixin-а Enumerable. Но есть одна часто используемая операция над Array и Hash, которая реализуется в характерном для Ruby стиле — это итерация по всем элементам коллекции. В Ruby итерация выполняется, как правило, с помощью метода each (для Hash также существует и each_pair):

```
[ 1, 'one', :first ].each do |item| puts item end
{ 1 => 'first', 2 => 'second' }.each do |p| puts "#{p[0]}:#{p[1]}" end
{ :first => 1, :second => 2 }.each_pair do |k, v| puts "#{k}:#{v}" end
```

Время от времени в Ruby-программе приходится определять литералы для Array, в котором содержатся только строки, например:

```
keywords = [ 'for', 'in', 'if', 'else', 'begin', 'end' ]
```

Чтобы упростить их запись, в Ruby существует еще два способа записи Array-литералов — %w и %W:

```
keywords = %w{for in if else begin end}
```

Различие между %w и %W такое же, как между %q и %Q для строковых литералов: в случае %w Ruby не вычисляет значений, заданных в формате #:

```
value = 1
a = %w{ #{value} }                  # => ["\#{value}"]
b = %W{ #{value} }                  # => ["1"]
```

4.13 Вызов методов и типы параметров

Одной из самых ярких граней Ruby является удобство создания встроенных мини-языков (т.н. embedded Domain-Specific Languages (DSL) [25], [26]), когда для какой-то предметной области

создается специализированный язык на основе Ruby. И программы на этом языке выглядят так, как будто они используют новые синтаксические конструкции, специально добавленные в язык. Пример такого специализированного мини-языка можно найти уже в стандартной библиотеке Ruby:

```
class ConnectionParams
  attr_accessor :host, :user, :password
  ...
end
```

Кажется, что конструкция attr_accessor — это особая синтаксическая конструкция языка, предназначенная только для описания getter/setter-ов. Еще дальше DSL-строение в Ruby продвинулось в таких знаковых для Ruby инструментах, как Ruby-On-Rails [5] и Rake (аналог утилиты make, но в синтаксисе Ruby, [27]).

Особенность Ruby в том, что все эти DSL на самом деле являются обычными Ruby-программами в *стандартном* синтаксисе Ruby. Т.е., никакой модификации языка не производится, просто используются его возможности. И одной из самых важных особенностей синтаксиса и семантики Ruby, непосредственно влияющих на создание DSL, является синтаксис вызова методов, а также типы параметров методов в Ruby.

ПРИМЕЧАНИЕ

Описываемые ниже возможности Ruby относятся к текущей стабильной ветке языка 1.8 (в частности, к 1.8.5). В следующей версии 1.9.* планируются некоторые изменения в этой области (см.например, [28]).

4.13.1 Необязательные скобки

Пожалуй, самой первой и самой запоминающейся особенностью синтаксиса Ruby являются необязательные скобки при вызове методов. Так, запись

```
attr_accessor :host, :user, :password
```

является эквивалентом более привычной записи:

```
attr_accessor(:host, :user, :password)
```

Опускание скобок при вызове функций является обычной практикой в Ruby. Существуют сложившиеся конструкции, в которых скобки традиционно не применяются, например, при обращении к методу puts или при итерации по коллекции с помощью метода each.

Естественно, не всегда Ruby способен правильно определить, к какому методу относятся те или иные аргументы, особенно при вложенных вызовах:

```
def meth_a(a, b)
end

def meth_b(a, b, c)
end

meth_b 1, meth_a 2, 3, 4
```

что приводит к сообщениям об ошибках, например:

```
-:7: parse error, unexpected tINTEGER, expecting kDO or '{' or '('
meth_b 1, meth_a 2, 3, 4
                  ^
```

В таких случаях скобки следует расставлять явно:

```
meth_b 1, meth_a(2, 3), 4
```

Как и любое сильнодействующее средство, необязательность скобок можно использовать не только во благо, но и во вред. Чрезмерное использование этой возможности очень сильно снижает читаемость кода, что особенно критично при сопровождении больших Ruby-программ. Поэтому имеет смысл ограничить использование необязательных скобок рамками DSL и традиционных идиом языка.

4.13.2 Значения параметров по умолчанию

В Ruby версии 1.8 некоторым параметрам метода можно присвоить значения по умолчанию. Как и в C++, эти параметры должны идти последними в списке параметров:

```
class ConnectionParams
  ...
  def initialize(
      host, user, password, options = {})
    ...
  end
  ...
end
```

Здесь параметр options имеет значение по умолчанию — пустой объект Hash. Поэтому параметр options при вызове метода может быть опущен. В этом случае для него будет использоваться значение по умолчанию:

```
# Используется значение по умолчанию.
params = ConnectionParams.new('server:3000', 'admin', 'secret')

# Значение для options задается явно.
params = ConnectionParams.new('server:3000', 'admin', 'secret',
    { :io_type => :nonblocking, :timeout => 300 })
```

Значения по умолчанию важны еще и потому, что в Ruby (в отличие от C++ или Java) нет перегрузки функций на основании списков параметров. Т.е. нельзя определить несколько методов с одним именем и разными списками аргументов, за выбор самого подходящего из которых отвечал бы интерпретатор. Вместо этого можно всего лишь назначать некоторым параметрам значения по умолчанию и при вызове метода опускать часть из них.

4.13.3 Параметры Array

В Ruby есть возможность пометить последний параметр метода знаком «звездочка»:

```
def method(*args)
...
end
```

что указывает Ruby специальным образом обрабатывать аргументы такого метода при вызове. Звездочка перед именем параметра означает, что параметр будет иметь тип Array, но при вызове метода аргументы будут задаваться в виде последовательности значений. Эта последовательность будет автоматически преобразована в объект Array, и этот объект будет передан в метод в виде одного аргумента. Например:

```
def method(*args)
  puts args.join(', ')
end

method(1, '2', :three, 4)       # => 1, 2, three, 4
```

Или, с учетом необязательности скобок:

```
method 1, '2', :three, 4
```

Именно за счет таких параметров работают многие DSL-образующие методы в Ruby. Например, упоминавшийся выше метод attr_accessor всего лишь вызывает для каждого своего аргумента стандартный метод attr. Простейшая реализация attr_accessor имеет вид:

```
class Object
  ...
  def Object.attr_accessor(*attrs)
    attrs.each do |a| attr(a, true) end
  end
  ...
end
```

Нужно отметить, что запись **something* является стандартной конструкцией Ruby, означающей «разворачивание» объекта Array в последовательность содержащихся в нем элементов, что часто используется в *параллельных присваиваниях*. Например:

```
a = [1, 2, 3, 4]
b, c, d = *a
b                                   # => 1
c                                   # => 2
d                                   # => 3
```

Здесь первые три значения из массива a были присвоены трем переменным-приемникам. Четвертое значение было проигнорировано, поскольку приемников было всего три. Если нужно, чтобы переменная d получила все оставшиеся в a значения, то следует использовать запись *d:

```
a = [1, 2, 3, 4]
b, c, *d = *a
b                                   # => 1
c                                   # => 2
d                                   # => [3, 4]
```

Поэтому запись параметра Array в виде **args* согласуется с операцией «разворачивания» значения Array посредством операции «звездочка», что также проявляется при необходимости передачи параметру Array значений из какого-то вектора:

```
a = [ 1, 2, 3, 4, 5 ]
method *a
```

Например, это может быть необходимо, если параметр Array нужно передать в другой метод в качестве аналогичного параметра Array:

```
def method(*args)
  puts *args
end
```

Параметры Array оказываются полезными еще в нескольких случаях. Например, при перекрытии метода базового класса, когда список параметров метода не важен. Скажем, при создании собственных классов в unit-тестах иногда бывает необходимо определить собственный конструктор. Но его формат должен в точности совпадать с форматом конструктора базового типа TestCase из стандартной библиотеки, поскольку объекты unit-тесты создаются не программистом, а фреймворком. Чтобы не зависеть от формата конструктора базового типа TestCase стандартной библиотеки, можно использовать параметр Array:

```
class TC_MyUnitTest < Test::Unit::TestCase
  # Неважно, какие аргументы передаются в конструктор unit-теста.
  # Пусть они выглядят как параметр Array.
  def initialize(*args)
    # Конструктор базового типа получает все аргументы в исходном виде.
    super(*args)
    ...
  end
  ...
end
```

Еще одним случаем использования параметров Array являются методы вроде send и method_missing, которым приходится передавать неизвестные изначально списки аргументов. Поэтому последним параметром этих методов является именно параметр Array:

```
# obj.send(symbol [, args...]) => obj
#
# Создание объекта типа Range путем ручной отправки сообщения :new
# классу Range со всеми необходимыми параметрами.
r = Range.send(:new, 1, 4, true)    # => 1...4
```

4.13.4 Параметры Hash

Время от времени в Ruby-коде можно встретить конструкции вида:

```
some_obj.some_method(:param => value, :another_param => another_value)
```

что выглядит как использование *именованных параметров* (как, например, в языке Ada). Но в Ruby нет именованных параметров. Вместо них используется параметр Hash — когда Ruby при вызове метода обнаруживает, что последняя часть списка аргументов записана в синтаксисе литерала Hash, то Ruby группирует эти элементы в один аргумент типа Hash:

```
def method(name, options)
  p name
  p options
end

method 'sample', :first => 1, :second => 2, :third => 3
```

что дает в результате:

```
"sample"
{:second=>2, :third=>3, :first=>1}
```

Гибкость синтаксиса Ruby проявляется здесь в том, что значения параметра Hash не нужно заключать в фигурные скобки при вызове метода (как это требуется для литерала Hash). Т.е. приведенная выше запись полностью эквивалентна непосредственному использованию литерала Hash:

```
method 'sample', { :first => 1, :second => 2, :third => 3 }
```

Такая гибкость в сочетании с необязательностью скобок и наличием параметров со значениями по умолчанию делает возможным запись вызовов методов, похожую на использование новых синтаксических конструкций. Например, следующий пример правил компиляции C-кода для Rake является обычной Ruby-программой:

```
file 'main.o' => ["main.c", "greet.h"] do
  sh "cc -c -o main.o main.c"
end
```

```
file 'greet.o' => ['greet.c'] do
  sh "cc -c -o greet.o greet.c"
end

file "hello" => ["main.o", "greet.o"] do
  sh "cc -o hello main.o greet.o"
end
```

в которой конструкция:

```
file 'main.o' => [ "main.c", "greet.h" ]
```

это записанный с использованием гибкости Ruby синтаксиса вызов метода file с единственным параметром Hash. Его можно было бы переписать вот так:

```
file({ 'main.o' => [ "main.c", "greet.h" ] })
```

Параметры Hash имеют очень большое значение, и их полезность сложно переоценить. В отличие от именованных аргументов, параметры Hash допускают расширение списка аргументов. Например, пусть базовый класс ConnectionParams получает свои параметры в виде одного параметра Hash и сохраняет их в специальном атрибуте:

```
class ConnectionParams
  def initialize(args)
    @args = args
  end

  def host; @args[ :host ]; end
  def user; @args[ :user ]; end
  def password; @args[ :password ]; end
end
```

что позволяет инициализировать объекты ConnectionParams следующим образом:

```
params = ConnectionParams.new :host => 'server:3000', :user => 'admin'
```

В производных классах список аргументов может быть расширен, например:

```
class NonBlockingConnection < ConnectionParams
  def initialize(args)
    # Если в списке параметров аргументы :io_type и :timeout не были
    # заданы явно, то они получают значения по умолчанию при помощи
    # метода Hash#merge.
    super({ :io_type => :nonblocking, :timeout => 300 }.merge(args))
    ...
  end
  ...
end

params = NonBlockingConnection.new :host => 'server:3000', :user => 'test'
params = NonBlockingConnection.new :host => 'server:3000', :user => 'test',
    :timeout => 1800

class DebugConnection < ConnectionParams
  def initialize(args = {})
    # Для тестового подключения используются одни и те же значения,
    # если только они не были заданы явно.
    super({ :host => 'dev.server:3000',
```

```
        :user => 'test',
        :password => 'test-password'
        :debug => true,
        :verbose => true }.merge(args))
  end
  ...
end

params = DebugConnection.new
params = DebugConnection.new :verbose => false
```

Также параметры Hash нивелируют отсутствие в Ruby возможности перегрузки методов. Достаточно определить один метод, который получает параметр Hash и разбирается, какие аргументы заданы, и что с ними нужно делать, как, например, метод find в ActiveRecord из Ruby-On-Rails:

```
# Описание отношения "Заказы" в качестве модели в RoR приложении.
class Order < ActiveRecord
  ...
end

# Поиск заказа с идентификатором 27.
o = Order.find(27)

# Поиск всех существующих заказов.
o = Order.find(:all)

# Поиск всех существующих заказов с условием.
o = Order.find(:all,
               :conditions => "name = 'dave' and pay_type = 'po'")

# Поиск с более сложным условием.
o = Order.find(:all,
               :conditions => [ "name = :name and pay_type = :pay_type",
                              {:pay_type => pay_type, :name => name} ])

# Поиск с условием, упорядочением и ограничением на количество
# найденных записей.
o = Order.find(:all,
               :conditions => "name = 'Dave'",
               :order => "pay_type, shipped_at DESC",
               :limit => 10)
```

Естественно, что такая гибкость при использовании параметра Hash не дается бесплатно. Во-первых, на программиста ложится ответственность по контролю за наличием обязательных ключей в Hash и их значениями. Во-вторых, при сложных сочетаниях ключей в Hash реализация метода может оказаться нетривиальной, что способно серьезно затруднить сопровождение кода. В-третьих, постоянное обращение к Hash за значениями аргументов может снизить производительность. Такова цена параметра Hash.

4.14 Блоки кода и объекты Proc

4.14.1 Общие сведения о блоках кода

Блок кода — это последовательность Ruby-выражений, заключенная в операторные скобки do/end или {}. Блок должен начинаться на той же строке, в которой происходит вызов некоторого метода, которому он предназначен. Блок кода может иметь необязательный список параметров, которые описываются в начале блока:

```
<вызов> do |p1, p2, ...| ... end
<вызов> { |p1, p2, ...| ... }
```

В самом простом случае блок кода используется в сочетании с методом, который нуждается в блоке кода. Например, метод Array#each:

```
[ 1, 2, 3 ].each do |i| puts i end
```

В данном случае метод each получает в качестве неявного аргумента блок кода, заключенный в операторные скобки do/end.

Существует два основных способа передачи блока кода в метод. В первом из них сигнатура метода никак не отражает того факта, что метод нуждается в блоке кода. А внутри метода управление в блок кода передается с помощью ключевого слова *yield*:

```
def n_times(n)
  for i in 1..n
    yield
  end
end

n_times(2) { print "Hello! " }  # => 'Hello! Hello! '
```

Если в блок нужно передать аргументы, то они указываются при обращении к yield:

```
def n_2_m(n, m)
  for i in n..m
    yield(i)
  end
end

n_2_m(3, 5) { |i| print "#{i}..." }   # => '3...4...5...'
```

Поскольку в сигнатуре метода никак не отражается факт ожидания методом блока кода, легко можно вызвать метод без передачи ему блока кода. Что вызовет исключение во время обращения к yield. Но наличие переданного блока кода можно проверить при помощи метода *block_given?*. Например:

```
# Конструирует параметры подключения к удаленному узлу.
# Если передан опциональный блок кода, то перед возвратом
# передает управление блоку для уточнения созданных параметров.
def make_default_connection_params
  defaults = load_defaults
  params = ConnectionParams.new(defaults)
  if block_given?
    yield(params)
  end
  params
end

# Получение параметров без их уточнения.
params = make_default_connection_params

# Получение параметров с назначением тестовых имени пользователя и
# пароля, если эти значения не были заданы по умолчанию.
params = make_default_condition_params do |p|
  if !p.user
    p.user = 'test'
    p.password = 'test-pwd'
  end
end
```

Второй способ передачи блоков кода требует, чтобы последний параметр метода был помечен символом амперсанда:

```
def n_times(n, &blk) ... end
```

В этом случае передаваемый блок кода автоматически конвертируется в объект стандартного класса Proc. Запуск блока кода на выполнение производится обращением к методу call этого объекта:

```
def n_2_m(n, m, &blk)
  for i in n..m
    blk.call(i)
  end
end

n_2_m(3, 5) { |i| print "#{i}..." }    # => '3...4...5...'
```

Если блок кода при вызове метода опущен, то помеченный амперсандом параметр получает значение nil:

```
def make_default_connection_params(&init)
  defaults = load_defaults
  params = ConnectionParams.new(defaults)
  if init
    init.call(params)
  end
  params
end
```

Главное различие между этими двумя подходами состоит в том, что при неявной передаче блока кода в метод нет возможности сохранить переданный блок и как-либо использовать его в дальнейшем. Можно только передать ему управление с помощью yield. В случае же с объектом Proc происходит работа с обычным объектом Ruby, который можно сохранить (что активно применяется при реализации обратных вызовов), можно передать параметром в другой метод, можно возвратить из метода. Строго говоря, на уровне интерпретатора Ruby блоки кода и объекты Proc являются разными сущностями. Но при их обычном использовании особой разницы между ними не заметно, поэтому часто можно считать, что блоки кода и объекты Proc — это одно и то же.

Получить объект Proc можно посредством конструктора класса Proc или с помощью стандартных методов Kernel#proc/Kernel#lambda:

```
hello = Proc.new { puts 'Hello' }
bye = lambda { puts 'Bye' }
```

которые затем можно передавать в качестве блоков кода при вызове методов:

```
def say_something
  yield
end
say_something &hello              # => 'Hello'
say_something &bye                # => 'Bye'
```

Амперсанд перед именами переменных с объектами Proc здесь необходим для того, чтобы интерпретатор Ruby передавал объекты в метод не как обычные аргументы, а именно как блоки кода.

Поскольку объекты Proc являются обычными объектами, то их можно передавать в методы и как обычные параметры:

```
# Устанавливает соединение, используя указанные параметры.
# Если в options есть ключ :pre_connect, то соответствующий ему
# блок кода вызывается до установления соединения. Если есть ключ
# :post_connect, то соответствующий ему блок кода вызывается сразу
```

```
# после установления соединения.
def connect(params, options = {})
  if nil != (pre_block = options.fetch(:pre_connect, nil))
    pre_block.call(params)
  end
  io = do_connect(params)
  if nil != (post_block = options.fetch(:post_connect, nil))
    post_block.call(io)
  end
  io
end

# Вызов без дополнительных обработчиков.
connect(make_connection_params)

# Вызов с pre connect обработчиком.
connect(make_connection_params,
    :pre_connect => proc { puts 'pre connect' })

# Вызов с pre/post connect обработчиками.
connect(make_connection_params,
    :pre_connect => proc { puts 'pre connect' },
    :post_connect => proc { puts 'post connect' })
```

Различие между операторными скобками do/end и {} состоит в том, что у do/end приоритет меньше, то есть блок do/end будет относиться ко всему выражению, а не к отдельной его части:

```
Ip = Struct.new(:host, :port)
ips = [ Ip.new('localhost', 3000), Ip.new('google.com', 80) ]
puts ips.inject('') do |r, ip| r << "#{ip.host}:#{ip.port}, "; r end
```

этот фрагмент завершается сообщением об ошибке:

```
-:3:in `inject': no block given (LocalJumpError)
        from -:3:in `each'
        from -:3:in `inject'
        from -:3
```

поскольку блок do/end был передан не в метод inject, а в метод puts. Чтобы блок кода относился к методу inject, его нужно заключить в {}:

```
puts ips.inject('') { |r, ip| r << "#{ip.host}:#{ip.port}, "; r }
```

Но, как правило, выбор между do/end и {} делается на основании эстетических соображений — короткие блоки кода, умещающиеся на одну строку, хорошо выглядят в {}. Более длинные блоки кода, занимающие несколько строк, лучше воспринимаются в скобках do/end.

Блоки кода сохраняют связь с контекстом, в котором они созданы. В этом плане блоки кода могут рассматриваться как *замыкания* (closures):

```
def consumer
  a = 3
  b = 4
  yield
end

def producer(&blk)
  a = 'hello'
  consumer do
    puts a + blk.call
```

```
  end
end

b = ', world'
producer { b }              # => 'hello, world'
```

Данный пример показывает, что блок, переданный в producer, сохраняет привязку к переменной b. В свою очередь блок, созданный в producer и переданный в consumer, сохраняет привязку к локальной переменной a, и поэтому локальные переменные a, b в методе consumer не оказывают на блок никакого воздействия. Это свойство блоков кода и объектов Proc активно используется, например, при реализации обратных вызовов.

Приведенный выше пример также демонстрирует, что блоки кода возвращают значение в вызвавший их метод. Таким возвращаемым значением, как и в случае обычных методов, является значение последнего выполненного в блоке выражения. Конструкция return также может использоваться в блоках кода, но если блок кода был создан посредством Proc::new, то return производит возврат не только из блока кода, но и из использующего блок кода метода. В то же время, если блок создается посредством методов proc/lambda, то return выполняет возврат только из блока кода:

```
def sample1
  (1..5).each do |v| return v end
  "done"
end

def sample2
  p = Proc.new do return 2 end
  p.call
  "done"
end

def sample3
  p = lambda do return 3 end
  p.call
  "done"
end

sample1                     # => 1
sample2                     # => 2
sample3                     # => "done"
```

Еще одно различие между блоками кода, созданными через Proc.new (или неявной передачей блока кода в метод) и методами proc/lambda заключается в проверке количества аргументов. Объекты Proc не проверяют количество аргументов:

```
nonchecked = Proc.new { |a, b, c| [ a, b, c ] }
checked = lambda { |a, b, c| [ a, b, c ] }

nonchecked.call(1, 2)         # => [1, 2, nil]
nonchecked.call(1, 2, 3, 4)   # => [1, 2, 3]
checked.call(1, 2)
```

В последнем случае лишние аргументы были просто проигнорированы. Но если вызвать call для checked с неверным количеством аргументов, то возникнет исключение:

```
-:2: wrong number of arguments (2 for 3) (ArgumentError)
        from -:6:in `call'
        from -:6
```

4.14.2 Блоки в качестве итераторов

Пожалуй, самым распространенным способом использования блоков кода является применение их в качестве итераторов в различных коллекциях. Самым простым примером является метод Array#each:

```
[ 1, 2, 3 ].each { |i| print i+1, ' ' }  # => 3 4 5
```

Эта же операция на C++ могла бы выглядеть следующим образом:

```
typedef std::vector< int > Vector;
Vector v;
// ... Инициализация ...
for(Vector::iterator it = v.begin(), end = v.end(); it != end; ++it)
  std::cout << (*it) + 1 << ' ';
```

Эти два примера показывают, что в Ruby стандартным подходом к итерациям является использование *внутреннего* итератора. В то же время другие языки, в частности C++ и Java, используют *внешние* итераторы. Хотя здесь правильнее говорить о доминирующих подходах, поскольку C++ и Java также содержат реализации итераций с использованием внутренних итераторов (std::for_each в C++, цикл foreach в Java). А Ruby поддерживает работу с внешними итераторами:

```
for i in [ 1, 2, 3 ]
  print i+1, ' '
end
```

Но, как правило, итерации с внутренними итераторами оказываются более компактными, хотя и требуется некоторое время, чтобы привыкнуть к некоторым операциям. Например, метод Enumerable#inject позволяет получить результат обработки всех элементов коллекции:

```
s = [ 1, 2, 3 ].inject('') { |r, i| r << "<#{i}>" }
s                                  # => "<1><2><3>"
```

что можно переписать с использованием for следующим образом:

```
s = ''
for i in [ 1, 2, 3 ]
  s << "<#{i}>"
end
s                                  # => "<1><2><3>"
```

Особенно удобно использование блоков кода в качестве итераторов, когда требуется выполнить какое-либо преобразование контейнера внутри выражения. Например, в следующем фрагменте из списка найденных файлов изымаются все файлы, в именах которых есть подстрока fragment_:

```
# Печать списка *.rb файлов за исключением некоторых вспомогательных файлов.
puts Dir[ '**/*.rb' ].delete_if { |n| n =~ /fragment_/ }.sort.join("\n")
```

4.14.3 Блоки и захват/освобождение ресурсов

Интересным примером использования блоков кода является сокрытие с их помощью операций захвата и освобождения каких-либо ресурсов. Например, если нужно прочитать что-нибудь из файла, то файл требуется открыть, выполнить чтение и закрыть файл. Но, поскольку во время чтения файла могут возникнуть какие-либо исключения, нужно предусмотреть закрытие файла в этом случае. В результате операция чтения будет выглядеть, например, так:

```
f = File.open(file_name, 'r')
begin
  ... # Чтение.
ensure
  # Сюда управление попадает при выходе из блока begin/end при
  # любом исходе, как в результате исключения, так и без него.
  # Конструкция ensure является аналогом finally в других языках.
  f.close
end
```

Однако такая работа с файлом не очень надежна, т.к. легко забыть написать конструкцию ensure. Кроме того, ее придется писать каждый раз при необходимости выполнения с файлом подобных действий. Но операцию закрытия файла можно сделать автоматической, если поместить все действия по работе с файлом в блок кода, который передается методу File::open:

```
File.open(file_name, 'r') do |file|
  ... # Чтение.
end
```

При этом сама реализация метода File::open выглядит приблизительно следующим образом:

```
class File
  def File.open(*args)
    result = f = File.new(*args)
    if block_given?
      begin
        result = yield f
      ensure
        f.close
      end
    end
    return result
  end
end
```

Таким образом, сначала создается объект типа File, в конструкторе которого выполняется открытие файла. Затем, если в метод open передан блок кода, то этот блок выполняется в рамках блока begin/end. По какой бы причине ни произошел возврат из блока кода, секция ensure закрывает открытый файл. Результирующим значением File::open при наличии блока кода является возвращенное из блока кода значение. Если же блок кода не задан, то возвращается объект типа File, а ответственность за закрытие файла ложится на программиста.

Подобный подход к захвату и освобождению ресурсов в стандартной библиотеке Ruby можно увидеть также при работе с многопоточностью: тело нити представляет собой блок кода, переданный в метод Thread::new или Thread::start:

```
# Параллельная запись файлов в разных нитях.
FILES = { 'a.tmp' => 'Content of file A',
    'b.tmp' => 'Content of file B',
    'c.tmp' => 'Content of file C' }
threads = FILES.map do |pair|
  Thread.new(*pair) do |name, content|
    File.open(name, 'w') do |file|
      file << content
    end
  end
end

threads.each do |thread| thread.join end
```

В случае с Thread::new/Thread::start создание новой нити и ее завершение (которые можно рассматривать как захват и освобождение ресурсов) выполняется внутри метода класса Thread, а прикладная логика — в переданном блоке кода.

4.14.4 Блоки и обратные вызовы

Блоки кода активно используются для реализации обратных вызовов — когда блок кода вызывается внутри некоторого фреймворка для обработки тех или иных событий. При этом сказывается еще и то, что блоки кода являются замыканиями и сохраняют связь с контекстом, в котором они были определены.

В качестве простого примера использования блоков кода для реализации обратных вызовов можно взять стандартную функцию *Kernel#at_exit*, которая позволяет выполнить некоторые действия при завершении программы. Приведенный ниже фрагмент создает временные файлы с параметрами запуска компилятора C++, после чего возвращает их имена. При этом для каждого файла регистрируется процедура его удаления при выходе — таким образом временные файлы будут автоматически удаляться при завершении программы:

```
def make_response_files(source_files)
  source_files.inject([]) do |result, src|
    name = "#{src}.#{Process.pid}.tmp"
    File.open(name, 'w') { |file| file << "cl -c -O2 #{src}" }
    at_exit { File.delete name }

    result << name
  end
end

response_files = make_response_files([ 'hello.cpp', 'world.cpp' ])
```

Этот простой пример показывает, что блок кода, будучи замыканием, сохраняет работоспособность даже после того, как создавший его контекст перестал существовать (метод make_response_files уже давно отработал).

Еще больше примеров использования блоков кода для реализации обратных вызовов можно увидеть в GUI-библиотеках для Ruby (например, FXRuby [30] и Tk [31]) — с помощью блоков кода оформляются реакции на нажатия горячих клавиш, воздействия на элементы управления и т.д. Например, вот так выглядит простая реализация программы HelloWorld с использованием FXRuby:

```
require 'fox16'

include Fox

theApp = FXApp.new

theMainWindow = FXMainWindow.new(theApp, "Hello")
theButton = FXButton.new(theMainWindow, "Hello, World!")
theButton.connect(SEL_COMMAND) { exit }

theApp.create
theMainWindow.show
theApp.run
```

В ней для кнопки «Hello, World» назначается блок кода с единственной операцией — завершением работы приложения при нажатии на кнопку.

4.14.5 Еще несколько общих слов

Блоки кода — это, без преувеличения, один из «краеугольных камней» языка Ruby. Они не очень сложны в освоении и очень удобны в использовании. Несколько основных сфер применения блоков кода в Ruby уже описано выше. Еще несколько, например, использование блоков в eval-методах и при динамическом расширении классов/объектов, описывается ниже.

4.15 Разнообразные eval-ы

В Ruby существует небольшое семейство методов, позволяющих динамически исполнять в программе новый код — eval-методы. Наиболее общий из них, *Kernel#eval*, получает строку, транслирует ее в Ruby-код, исполняет его и возвращает результат:

```
a = eval('(1+2)*3')              # => 9
```

Код, который генерируется внутри eval, как бы встраивается в место вызова eval. Это имеет большое значение, когда в аргументе eval определяются новые переменные, методы или классы:

```
a = calculate                        # -:1: undefined local variable or method
                                     # `calculate' for main:Object (NameError)
eval('def calculate; (1+2)*3; end')
a = calculate                        # => 9
```

Метод calculate был введен в программу во время работы метода eval.

Если eval получает один аргумент, то он работает в том контексте, в котором его вызвали. Но методу eval можно передать дополнительный аргумент, binding, который назначает eval контекст, в котором нужно работать:

```
class Test
  def initialize
    @greeting = 'Hello, World!'
  end
end

t = Test.new

eval('@greeting')                    # => nil
eval('@greeting', t.send(:binding))  # => "Hello, World!"
```

В первом eval получен nil, поскольку поиск атрибута @greeting производился в глобальном контексте. Естественно, что там его не было, но по правилам обращения к атрибутам в Ruby это не вызвало ошибки, а привело к возвращению значения nil. Во втором eval получено значение атрибута @greeting объекта t, поскольку eval выполнялся в контексте этого объекта.

В данном примере для получения контекста объекта t использовался метод *Kernel#binding*, который присутствует в каждом Ruby-объекте. Но обратиться к нему извне объекта нельзя, т.к. это приватный метод, и он может быть вызван только самим объектом. Поэтому для получения контекста можно было либо определить в классе Test вспомогательный метод:

```
class Test
  ...
  def get_binding
    binding
  end
end
```

либо, как было показано выше, воспользоваться возможностью отсылки объекту сообщения с помощью send.

Методы *Module#module_eval* и *Module#class_eval* являются синонимами (т.е. имеют одну реализацию) и исполняют переданный им код в контексте указанного класса/модуля. Обычно они используются для расширения класса новыми методами. Например, пусть в классе ProjectDescription требуется создать группу методов для добавления в проект различных сущностей (файлов, модулей, ресурсов, описаний и пр.). Причем для каждой сущности должно быть два метода: первый метод получает одно имя, а второй метод – список имен, чтобы можно было работать с ProjectDescription следующим образом:

```
prj = ProjectDescription.new('my.project')
prj.add_file 'my.rb'
prj.add_files Dir[ 'lib/**/*.rb' ]
prj.add_description 'README'
prj.add_descriptions Dir[ 'docs/**/*' ]
```

Одним из способов решения данной задачи является ручное создание всего лишь одного метода, скажем, получающего список имен:

```
class ProjectDescription
  def add_files(names); ... end
  def add_descriptions(names); ... end
  ...
end
```

а варианты методов с единственным именем в качестве аргумента генерируются на основе уже существующих методов:

```
class ProjectDescription
  ...
  # Вспомогательный метод для генерации.
  def ProjectDescription.define_singular_form_method(method)
    class_eval %Q{
      def #{method[0...-1]}(a)
        #{method} [a]
      end
    }
  end

  # Генерация недостающих методов.
  define_singular_form_method :add_files
  define_singular_form_method :add_descriptions
  ...
end
```

В такой реализации каждое обращение к define_singular_form_method будет генерировать еще один метод, принимающий в качестве аргумента единственное имя.

В отличие от eval, module_eval/class_eval могут получать в качестве аргумента не только строку, но и блок кода:

```
class ProjectDescription
  # Вспомогательный метод для генерации методов default_*.
  def ProjectDescription.define_default_value_getter(what, value)
    class_eval do
      define_method "default_#{what}" do
        value
      end
    end
  end

  define_default_value_getter :files, [ 'init.rb', 'shutdown.rb' ]
  define_default_value_getter :descriptions, [ 'README', 'LICENSE', 'NEWS' ]
end

prj = ProjectDescription.new
prj.default_files                # => ["init.rb", "shutdown.rb"]
prj.default_descriptions         # => ["README", "LICENSE", "NEWS"]
```

Блок кода здесь используется потому, что значения, которые должны возвращаться сгенерированными методами default_*, присутствуют в программе в виде Ruby-объектов, а не строк.

Последним методом семейства eval является метод *Object#instance_eval*. Как следует из его названия, instance_eval выполняет исполнение переданного ему кода в контексте указанного объекта:

```
'Hello!'.instance_eval('length')    # => 6
```

Так же, как и module_eval/class_eval, instance_eval может получать в качестве аргумента блок кода:

```
'Hello!'.instance_eval { length }    # => 6
```

Последний вариант instance_eval демонстрирует замечательную особенность Ruby, которая делает возможным создание конструкций, выглядящих как декларативные описания DSL. Например, описание:

```
project 'My' do
  files default_files + [ 'main.rb', 'help'.rb' ]
  descriptions default_descriptions + [ 'INSTALL', 'BUGS', 'ChangeLog' ]
  resources Dir[ 'images/**/*.png' ]
end
```

выглядит как какой-нибудь конфигурационный файл с похожим на Ruby синтаксисом. Но в действительности это всего лишь фрагмент Ruby-программы, использующей такие свойства Ruby, как необязательность скобок при вызове методов, передача блоков в качестве параметров и instance_eval.

Чтобы продемонстрировать, как это работает, сначала нужно определить реальный класс ProjectDescription, а затем вспомогательный метод project:

```
class ProjectDescription
  def initialize(name)
    @name = name
    @files = []
    @descriptions = []
    @resources = []
  end

  def files(names); @files += names; end
  def descriptions(names); @descriptions += names; end
  def resources(names); @resources += names; end

  def ProjectDescription.define_default_value_getter(what, value)
    class_eval do
      define_method "default_#{what}" do
        value
      end
    end
  end

  define_default_value_getter :files, [ 'init.rb', 'shutdown.rb' ]
  define_default_value_getter :descriptions, [ 'README', 'LICENSE', 'NEWS' ]
  define_default_value_getter :resources, [ 'logo.png' ]
end

def project(name, &init)
  p = ProjectDescription.new(name)
  p.instance_eval &init
  p
end

prj = project 'My' do
  files default_files + [ 'main.rb', 'help.rb' ]
  descriptions default_descriptions + [ 'INSTALL', 'BUGS', 'ChangeLog' ]
```

```
  resources Dir[ 'images/**/*.png' ]
end
prj                             # => #<ProjectDescription:0x27f0538
                                      @resources=[],
                                      @files=["init.rb", "shutdown.rb",
                                              "main.rb", "help.rb"],
                                      @name="My",
                                      @descriptions=["README", "LICENSE",
                                                     "NEWS", "INSTALL",
                                                     "BUGS", "ChangeLog"]>
```

Вся магия скрывается в реализации метода project. В нем сначала создается объект класса ProjectDescription, а затем в контексте этого объекта выполняется блок кода. Эффект от instance_eval получается такой, как будто для объекта был вызван какой-то его собственный метод:

```
class ProjectDescription
  ...
  def initialize_my_project
    files(default_files() + [ 'main.rb', 'help.rb' ])
    descriptions(default_descriptions() + ['INSTALL', 'BUGS', 'ChangeLog'])
    resources(Dir[ 'images/**/*.png' ])
  end
end
p = ProjectDescription.new('My')
p.initialize_my_project
```

4.16 Расширение уже существующих классов и объектов

В отличие от многих языков программирования, классы в Ruby являются *открытыми*. Как и пространства имен в C++, классы в Ruby можно дополнять новыми атрибутами/методами практически в любое время и в любом месте:

```
class ProjectDescription
  ...
  def files; ...; end
  def descriptions; ...; end
  def resources; ...; end
end
...
# Где-то, возможно даже в другом rb-файле.
class ProjectDescription
  ...
  def pictures; ...; end
  def audios; ...; end
  def videos; ...; end
end
...
# Еще где-то.
class ProjectDescription
  ...
  def presentations; ...; end
end
```

Такое расширение уже существующих классов может оказаться востребованным в различных ситуациях. Например, когда реализация части методов зависит от платформы, на которой работает программа:

```
class Application
  def run
    load_configuration
    perform_work
    store_results
  end

  def perform_work; ...; end
  def store_results; ...; end
end

# Реализация метода load_configuration зависит от платформы.
if /mswin/ =~ Config::CONFIG[ 'host_os' ]
  require 'app-mswin'
else
  require 'app-non-mswin'
end

# Файл app-mswin.rb. Реализация загрузки конфигурации из реестра.
class Application
  def load_configuration; ...; end
end

# Файл app-non-mswin.rb. Реализация загрузки конфигурации из файлов.
class Application
  def load_configuration; ...; end
end
```

При расширении класса автоматически расширяются все уже созданные объекты этого класса. То есть, если в класс были добавлены новые методы, они автоматически появляются в каждом из уже существующих объектов:

```
class Demo
  def first; end
  def second; end
end

a = Demo.new
a.class.instance_methods(false)     # => ["second", "first"]

class Demo
  def third; end
end

a.class.instance_methods(false)     # => ["second", "first", "third"]
```

Если в дополнительном определении класса заново реализуется уже имеющийся в классе метод, то данный метод получает новую реализацию у всех существующих объектов этого класса:

```
class Demo
  def greeting
    'hi'
  end
end

a = Demo.new
a.greeting                          # => "hi"

class Demo
  def greeting
    'Hello'
  end
end

a.greeting                          # => "Hello"
```

Расширять в Ruby можно любые классы, даже стандартные. Например, Ruby-On-Rails расширяет стандартные классы Integer, String и Time для предоставления удобных вспомогательных методов [19]:

```
puts 20.kilobytes                       # => 20480
puts 20.hours.from_now                  # => Wed May 11 13:03:43 CDT 2005
puts "cat".pluralize                    # => cats
puts "cats".singularize                 # => cat
puts Time.now.at_beginning_of_month     # => Sun May 01 00:00:00 CDT 2005
```

Все вышесказанное распространяется и на модули. Модули, как и классы, являются открытыми. Поэтому расширение модуля или смена реализации метода в модуле автоматически распространяется на объекты всех типов, в которых модуль был использован в качестве примеси:

```
module Greeting
  def hello; 'Hi!'; end
end

class Demo
  include Greeting
end

a = Demo.new
a.public_methods.grep(/(hello|applaud)/)    # => ["hello"]
a.hello                                     # => "Hi!"

# Теперь модуль Greeting модифицируется.
module Greeting
  def hello; 'Hello!'; end
  def applaud; 'Bravo!'; end
end

a.public_methods.grep(/(hello|applaud)/)    # => ["hello", "applaud"]
a.hello                                     # => "Hello!"
a.applaud                                   # => "Bravo!"
```

(метод *public_methods* по умолчанию возвращает список всех публичных методов объекта, включая унаследованные из базовых классов и примесей, но этот список большой, поэтому из него выделяются только методы с именами hello и applaud).

То, что в Ruby можно снабдить собственными методами любой, даже чужой, класс, открывает перед разработчиком новые возможности. Например, если при использовании сторонней библиотеки обнаруживается ошибка в ее реализации, то эта ошибка может быть исправлена без модификации исходного текста библиотеки — достаточно модифицировать проблемный класс в своем коде. Еще один пример: при использовании какого-то готового фреймворка может потребоваться модифицировать какой-либо его класс. Скажем, для того, чтобы снабдить его новой функциональностью или дополнительными атрибутами. Сделать это при помощи обычного наследования невозможно, если объекты нужного класса создаются где-то внутри фреймворка. Но можно расширить класс собственными методами/атрибутами, и тогда фреймворк автоматически начнет создавать объекты уже обновленного класса. Ruby-On-Rails демонстрирует это на примере стандартного класса Integer, объекты которого создаются где-то в глубине интерпретатора Ruby.

Открытыми в Ruby являются не только классы, но и объекты. Любой объект может быть расширен собственными методами, которые будут присутствовать только у него, но не у других объектов того же класса:

```
class Demo
  def f; end
  def g; end
  def j; end
end
```

```
a, b, c = Demo.new, Demo.new, Demo.new
a.public_methods(false)             # => ["g", "f", "j"]
b.public_methods(false)             # => ["g", "f", "j"]
c.public_methods(false)             # => ["g", "f", "j"]

# Добавление нового метода в объект a.
def a.k
end

# Добавление нового метода в объект b. Используется другой способ записи.
class <<b
  def m; end
end

a.public_methods(false)             # => ["g", "f", "j", "k"]
b.public_methods(false)             # => ["g", "f", "j", "m"]
c.public_methods(false)             # => ["g", "f", "j"]
```

(обращение к методу public_methods с аргументом false приводит к возврату списка только собственных методов объекта, без унаследованных из базовых классов и примесей).

Можно не только расширять объект новыми методами, но и менять реализацию существующих методов. Причем делать это не только для собственных объектов, но и для любых объектов, в том числе и для объектов стандартных типов:

```
a = [ 1, '1', :first ]
b = a.dup

a.to_s                              # => "11first"
b.to_s                              # => "11first"

class <<a
  def to_s
    r = map do |i| "#{i}:#{i.class.name}" end
    "[#{r.join(',')}]"
  end
end

a.to_s                              # => "[1:Fixnum,1:String,first:Symbol]"
b.to_s                              # => "11first"
```

Способ записи расширения объекта в виде конструкции class <<obj приоткрывает завесу над тем, каким образом в Ruby происходит расширение объекта. При расширении объекта Ruby создает экземпляр специального класса, в котором описываются отличия этого объекта от первоначального класса. Такой экземпляр называется *singleton-класс*, и, хотя и является обычным Ruby-классом, не имеет собственного имени. Характерная особенность singleton-класса состоит в том, что сколько бы раз объект не расширялся новыми методами, модифицируется всегда один и тот же экземпляр singleton-класса:

```
class Demo
  # Это должен быть идентификатор объекта с описанием класса Demo.
  self.object_id                    # => 20943230
end

a = Demo.new
# Это должен быть идентификатор объекта a.
a.object_id                         # => 20943200

class <<a
  def hello; puts 'hello'; end

  # Это должен быть идентификатор объекта с описанием singleton-класса
```

```
  # для объекта a.
  self.object_id                      # => 20942820
end

a.hello

class <<a
  def bye; puts 'bye'; end

  # Это должен быть идентификатор уже известного singleton-класса
  # для объекта a.
  self.object_id                      # => 20942820
end

a.bye
```

Данный пример показывает, что идентификатор singleton-класса для объекта a остается неизменным как при добавлении метода hello, так и при добавлении метода bye.

Тот факт, что singleton-класс всегда существует в единственном экземпляре, выглядит не столь серьезно, пока речь идет об обычных объектах. Но если вспомнить, что в Ruby все является объектом, и даже в описании класса есть self, который указывает на объект-описание этого класса, то оказывается, что singleton-классы можно создавать даже для классов:

```
class Demo
  # Это уже singleton-класс для класса Demo.
  class <<self
    def hello
      'Hello'
    end
  end

  hello                               # => "Hello"
end

Demo.hello                            # => "Hello"
```

Определение метода hello посредством singleton-класса для класса Demo эквивалентно определению обычного метода класса. Но все становится еще интереснее, если вспомнить, что любой класс может иметь переменные экземпляра (instance variable). Определение переменных экземпляра в singleton-классе класса Demo создает атрибуты класса Demo, которые принадлежат только классу Demo и никому больше (в то время как переменные класса (class variables) принадлежат также и производным классам). Что весьма важно при метапрограммировании в Ruby [32].

В Ruby можно выполнять подмешивание модулей не только к классам, но и к отдельным объектам. Здесь также используется singleton-класс объекта:

```
module Greeting
  def hello; 'Hi!'; end
end

class Demo
end

a = Demo.new
b = Demo.new
a.public_methods.grep(/(hello|applaud)/)    # => []
b.public_methods.grep(/(hello|applaud)/)    # => []

# Выполняем подмешивание модуля Greeting только к объекту a.
# Как вариант более компактной записи можно было бы использовать
```

```ruby
# метод Object#extend: a.extend(Greeting)
class <<a
  include Greeting
end

a.public_methods.grep(/(hello|applaud)/)    # => ["hello"]
b.public_methods.grep(/(hello|applaud)/)    # => []

a.hello                                       # => "Hi!"

# Теперь модуль Greeting модифицируется.
module Greeting
  def hello; 'Hello!';
  def applaud; 'Bravo!'; end
end

a.public_methods.grep(/(hello|applaud)/)    # => ["hello", "applaud"]
a.hello                                     # => "Hello!"
a.applaud                                   # => "Bravo!"
```

Возможность расширения классов и объектов в Ruby вызывает много споров. Кто-то справедливо считает, что необдуманное ее применение приведет к проблемам, особенно при сопровождении больших проектов. Тем не менее, эта возможность в Ruby есть. Выделяются, как минимум, две ситуации, в которых она будет востребована:

- смена реализации методов без останова приложения. Это важно для приложений, работающих в режиме 24x7, поскольку можно исправлять небольшие ошибки без перезапуска приложений.
- Наполнение специфической прикладной функциональностью чужих классов без модификации их исходных текстов (как делает Ruby-On-Rails со стандартными классами Integer, Time и String).

Как бы то ни было, динамическое расширение классов, модулей и объектов является одной из основополагающих возможностей языка. Поэтому при использовании Ruby нужно принимать ее во внимание, поскольку она действительно предоставляет уникальные возможности.

4.17 Переопределение методов

Как было показано выше, в Ruby можно переопределить метод объекта. Но не всегда требуется полностью менять реализацию метода, иногда необходимо дополнить уже существующий метод дополнительными действиями. Например, захватить какой-то ресурс в эксклюзивном режиме, выполнить действия метода, после чего освободить ресурс. В Ruby это выполняется посредством назначения псевдонима, осуществляемого с помощью *Method#alias_method*, с последующим переопределением метода:

```ruby
class Demo
  def do_something
    puts 'doing something'
  end
end

d = Demo.new
# Используется оригинальная версия метода do_something.
d.do_something

# Теперь объект d расширяется методами lock_resource, unlock_resource
# и получает новую реализацию метода do_something.
class <<d
  # Оригинальная версия do_something будет доступна под этим именем.
  alias_method :original_do_something, :do_something
  def do_something
    lock_resource
```

```
    original_do_something
  ensure
    unlock_resource
  end

private
  def lock_resource; puts 'locking'; end
  def unlock_resource; puts 'unlocking'; end
end

# Теперь будет работать новая версия do_something.
d.do_something
```

что дает при выполнении:

```
doing something
locking
doing something
unlocking
```

Принцип действия основывается на том, что после вызова alias_method для объекта d существует два метода с разными именами, но одинаковым телом. Переопределение do_something назначает новое тело для метода с именем do_something, но имя original_do_something продолжает ссылаться на старое тело метода.

Прием с alias_method используется не только для переопределения методов в конкретном объекте, но и для переопределения методов в классах. Ведь если метод переопределяется в результате наследования, то для обращения к реализации из базового класса достаточно указать ключевое слово super:

```
class Base
  def f; puts 'Base#f'; end
end

class Derived < Base
  def f
    puts 'Derived#f'
    super
  end
end

Derived.new.f                    # => Derived#f
                                 #    Base#f
```

Но если наследования нет, а класс расширяется новыми методами, то для обращения к уже существующей реализации метода приходится применять alias_method:

```
class SomeComplexClass
  ...
  def to_s
    # ...Сложное преобразование значения объекта в строку...
  end
end

# Если выполняется отладка, то часть методов SomeComplexClass должна
# быть снабжена дополнительной информацией.
if $DEBUG
  class SomeComplexClass
    alias_method :release_to_s, :to_s
    def to_s
      debug_info + release_to_s
    end
  end
end
```

Использование alias_method в сочетании с тем фактом, что в Ruby все методы являются виртуальными, может привести к неожиданным эффектам:

```
class Base
  def helper
    puts "Base#helper"
  end

  def action
    helper
  end
end

b = Base.new
b.action

class Derived < Base
  alias_method :base_helper, :helper
  def helper(what)
    puts "Derived#helper(#{what})"
  end
end

d = Derived.new
d.action
```

Здесь метод Base#helper вызывается из Base#action, но в производном классе Derived оригинальная версия helper переименовывается в base_helper. Новая же версия Derived#helper имеет другое количество параметров. Поэтому обращение к helper из Base#action приведет к ошибке:

```
Base#helper
-:7:in `helper': wrong number of arguments (0 for 1) (ArgumentError)
        from -:7:in `action'
        from -:22
```

Ошибка возникает из-за того, что в Base#action выполняется вызов метода без параметров с именем helper, но в объекте d этот метод уже требует одного параметра. Если бы в Ruby методы были не виртуальными, то в Base#action вызывалась бы версия Base#helper. Однако из-за виртуальности всех методов происходит вызов именно Derived#helper. Поэтому нужно внимательно следить за сигнатурами методов, переопределяемых с помощью alias_method.

4.18 method_missing

Когда интерпретатор Ruby определяет, что у некоторого объекта вызывается какой-то метод, он ищет реализацию этого метода в классе объекта, затем во всех базовых классах и примесях (включая и singleton-классы). Если метод с искомым именем не находится, у объекта вызывается специальный метод *method_missing*. По умолчанию его реализацией является реализация в *Kernel#method_missing*, которая порождает исключение NoMethodError. Однако method_missing можно перекрыть в своем классе или объекте.

Переопределение method_missing является довольно распространенным приемом, который можно использовать для разных целей. Например, с его помощью можно сократить объем кодирования getter/setter-ов для объектов, хранящих большое количество разнообразных значений:

```
class ValueHolder
  def initialize
    # Этот hash-map будет содержать все помещаемые в объект значения.
    @values = {}
  end

  def method_missing(symbol, *args)
```

```
    name = symbol.id2name  # Идентификатор метода преобразовывается в строку.
    if name =~ /(.+)=$/
      # Это setter в стандартном соглашении Ruby: a.b = x.
      @values[ $1 ] = *args
    else
      # Считаем, что происходит вызов getter-a: a.b.
      # Если значения для name еще не определено, то
      # будет возвращено nil.
      @values[ name ]
    end
  end
end

v = ValueHolder.new
v.a                                 # => nil
v.a = 'a'
v.a                                 # => "a"
v.something_else = :BlaBlaBla
v.something_else                    # => :BlaBlaBla
```

Показанный выше класс ValueHolder является примитивной демонстрацией принципа работы класса OpenStruct [37] из стандартной библиотеки Ruby, в основе которого также лежит использование method_missing.

Но наибольшее значение method_missing имеет для построения DSL – настолько важное, что даже в документации к Kernel#method_missing приводится пример класса для хранения римских чисел, в котором method_missing служит для разбора римской нотации:

```
class Roman
  def romanToInt(str)
    # ...
  end
  def method_missing(methId)
    str = methId.id2name
    romanToInt(str)
  end
end

r = Roman.new
r.iv                                #=> 4
r.xxiii                             #=> 23
r.mm                                #=> 2000
```

Одним из самых ярких и впечатляющих примеров использования method_missing для организации DSL является библиотека Builder [38], которая позволяет записывать XML/HTML-конструкции в виде Ruby-кода:

```
require 'rubygems'
require_gem 'builder', '~> 2.0'

builder = Builder::XmlMarkup.new
xml = builder.person { |b|
  b.name    "Jim"
  b.phone   "555-1234"
  b.address { |a|
    a.town   "New-York"
    a.zip    "655374"
    a.street "Broadway"
  }
}
xml # => "<person><name>Jim</name><phone>555-1234</phone>
          <address><town>New-York</town><zip>655374</zip>
          <street>Broadway</street></address></person>"
```

4.19 Утиная типизация

Еще одной, очень горячо обсуждаемой особенностью Ruby является т.н. «утиная типизация» (duck typing). Ее принцип состоит в том, что если некий объект «ходит как утка и крякает как утка» значит он — утка. Подобие объекта чему-либо проверяется наличием методов с нужными именами и сигнатурами, то есть наличие у объекта подходящего метода более важно, чем наследование класса объекта от некоторого базового класса.

Например, пусть требуется создать метод do_something_important, который протоколирует ход своей работы в журнальный файл. Объект файла-журнала должен передаваться методу в качестве параметра. Поскольку задача do_something_important заключается в выполнении собственных действий, то подготовка подходящих для этого условий – не его проблема:

```
def do_someting_important(log)
  log << 'loading configuration'
  load_config

  log << 'loading input data'
  load_input_data
  ...
end
```

Код метода do_something_important не зависит от того, является ли параметр log-файлом или еще чем-нибудь. Важно лишь наличие метода <<, который дописывает очередное сообщение в конец журнала. А это значит, что методу do_something_important можно передать в качестве параметра log *любой* объект, имеющий метод <<, например, объект класса String, тогда протоколирование будет вестись в строку. Или объект класса Array, тогда каждое новое сообщение будет дописываться в конец вектора.

Простейшим полезным проявлением утиной типизации в отношении метода do_something_important является создание unit-тестов для проверки его работы. Если бы do_something_important требовал работы именно с файлом, в unit-тесте пришлось бы каждый раз открывать файл, затем считывать его содержимое для сравнения с ожидаемым результатом, затем закрывать файл и удалять его. Но, поскольку do_something_important все равно, чем является его параметр log, то при тестировании do_something_important можно использовать, например, объект Array:

```
class TC_DoSomethingImportant < Test::Unit::TestCase
  def test_action_sequence
    log = []
    do_something_important(log)

    assert_equal(
        [ 'loading configuration', 'loading input data', ... ], log)
  end
  ...
end
```

Такая форма утиной типизации похожа на обобщенное (generic) программирование в таких статически типизированных языках, как Java, C# и C++/D. Но в Java и C# требуется указывать, какой интерфейс реализуют типы-параметры, в противном случае нельзя обращаться к методам объектов-параметров. Поэтому утиную типизацию в Ruby лучше сравнивать с шаблонами C++/D, в которых указание ограничивающих интерфейсов для типов-параметров не требуется, а проверка наличия нужных методов у объектов проверяется в момент компиляции программы.

Главным и принципиальным различием между утиной типизацией в Ruby и шаблонами в C++/D является то, что в C++/D компилятор гарантирует наличие нужных и совпадающих по сигнатуре параметров у объектов типов-параметров. Поскольку Ruby — это динамический язык, провести подобные проверки невозможно, и наличие методов у объектов проверяется непосредственно в момент выполнения.

Из этого следует еще одно отличие утиной типизации в Ruby от шаблонов C++/D: в статически типизированных языках для адаптации к особенностям шаблона необходимо прибегать к технике метапрограммирования. Например, чтобы во время компиляции C++-кода определить, есть ли у объекта метод swap, приходится использовать сложные шаблонные конструкции. А в Ruby подобная проверка выполняется во время исполнения посредством метода *Object#respond_to?*:

```
def swap_objects(a, b)
  if a.respond_to?(:swap)
    # Поскольку объект a поддерживает операцию swap, она используется
    # для обмена значениями объектов.
    a.swap(b)
  else
    # Операция swap не поддерживается, поэтому приходится реализовывать
    # ее самостоятельно.
    t = a.clone
    a = b
    b = t
  end
end
```

Метод respond_to? чрезвычайно важен при использовании утиной типизации, поскольку подобие объекта чему-либо проверяется по наличию у объекта необходимых методов, а не по принадлежности к какой-то иерархии классов. Например, метод *YAML::load* рассчитан на получение входного потока для разбора формата YAML [33] как в виде строки, так и в виде объекта ввода-вывода (базовым классом для которого в Ruby является класс IO [34]):

```
r = YAML.load("Host: localhost\nPort: 80")
r = File.open('info.yml', 'r') do |file| YAML.load(file) end
```

Простейшая реализация YAML::load могла бы содержать проверку типа аргумента:

```
def YAML.load(stream)
  if stream.kind_of?(String)
    # Передана строка, поэтому нужно преобразовать ее в поток.
    stream = StringIO.new(stream)
  elsif !stream.kind_of?(IO)
    # Невозможно работать с другим типом потока.
    ...
  end
  ...
end
```

Однако такая реализация как раз не поддерживает утиную типизацию, поскольку она принимает только наследников класса String (т.е. строки) или только наследников класса IO (т.е. потоки ввода-вывода). Но в программе могут быть и другие объекты, которые, не будучи наследниками String или IO, все равно смогут предоставить свое содержимое методу YAML::load. Например, объект, считывающий и записывающий информацию на внешнее устройство, вроде SmartCard. Для того чтобы этот объект мог выступать для YAML::load в качестве строки или потока ввода-вывода, можно использовать соглашения Ruby о преобразовании объектов. Эти соглашения реализуются в виде методов *to_** — to_s, to_str, to_i, to_int, to_io и т.д. Если объект поддерживает какой-то из них, значит, он поддерживает соответствующее преобразование. Скажем, если объект имеет метод to_io, то этот метод возвращает реализацию интерфейса IO. Если объект имеет метод to_s, то значение этого объекта может быть представлено в виде строки (аналог метода Object.toString в Java). А если объект имеет метод to_str, значит, объект может использоваться в любых операциях, в которых ожидается строка. Эти методы называются в Ruby протоколами преобразования (conversion protocols).

С использованием протоколов преобразования метод YAML::load может быть переписан так, чтобы действительно поддерживать утиную типизацию:

```
def YAML.load(stream)
  if stream.respond_to?(:to_str)
    # Объект может выступать в качестве строки.
    stream = StringIO.new(stream.to_str)
  elsif stream.respond_to?(:to_io)
    # Объект может выступать в качестве потока ввода-вывода.
    stream = stream.to_io
  else
    # Невозможно работать с другим типом потока.
    ...
  end
  ...
end

# Класс для работы со смарт-картами.
class SmartCard
  # Реализация интерфейса IO для выполнения операций ввода-вывода.
  class IOImplementation
    ...
  end
  ...
  def to_io
    IOImplementation.new(self)
  end
  ...
end

# Класс для работы с подписанными криптографическим образом данными.
class CryptographicalySignedData
  ...
  def to_str
    ... # Возвращение значения в качестве строки.
  end
  ...
end

YAML.load(SmartCard.new('PCSC-Reader-1'))
YAML.load(CryptographicalySignedData.new(some_crypto_device))
```

В таком виде объекты SmartCard и CryptographicalySignedData могут использоваться с методом YAML::load, поскольку они для YAML::load выглядят «как утки» — так же ходят и так же крякают.

4.19.1 Лирическое отступление: священные войны

Говоря об утиной типизации, невозможно не сказать о том, что утиная типизация является предметом ожесточенных споров между сторонниками статической и динамической типизации. Сторонники статической типизации настаивают на том, что компилятор предоставляет программистам больше гарантий корректности программы, поскольку на этапе компиляции проверяется, что обобщенные функции получают только аргументы ожидаемых ими типов. Соответственно, не имея таких гарантий, в динамически типизированном Ruby программист остается один на один со своими ошибками, и поэтому связанных с типизацией ошибок в Ruby-коде должно быть гораздо больше, чем в Java/C#/C++/D (здесь может быть подставлено любое другое имя).

Сторонники же динамической типизации в Ruby оперируют другим аргументом — из-за того, что нет статического контроля, программист освобождается от необходимости добавления в программу новых сущностей (интерфейсов для типов-параметров), которые нужны всего лишь для удовлетворения компилятора. Соответственно, уменьшение количества сущностей благоприятно сказывается на объеме и качестве программы, поскольку программист меньше отвлекается на второстепенные детали.

Кроме того, динамичность языка и отсутствие статического контроля означает, что в Ruby-программе, в принципе, в любом месте может оказаться объект не того типа. Поэтому для проверки корректности программы не остается других возможностей, кроме ее тестирования (в первую

очередь посредством unit-тестов). А тесты легко выявляют ошибки типизации, в том числе и связанные с утиной типизацией. Вероятно поэтому практикующие Ruby программисты просто не встречаются с тем количеством ошибок, которое предрекают им сторонники статической типизации.

К сожалению, критика утиной типизации в Ruby зачастую всплывает в контексте священных войн «статика vs динамика», где под понятие «динамически типизированных языков» автоматически попадают *все* динамически типизированные языки. Но эти языки очень разные и проблемы, которые есть в одних, полностью отсутствуют в других. Так, система типов в Ruby и Smalltalk серьезно отличается от таковой, например, в Perl, JavaScript или Visual Basic. Поэтому в Ruby напрочь отсутствуют такие страшилки, как операции арифметического сложения целого и строки, которыми очень любят пугать в подобных непрекращающихся спорах.

Такое пространное лирическое отступление было сделано специально, чтобы подчеркнуть очень важную вещь — опасения по поводу ошибок с типами при использовании утиной типизации могут присутствовать при выборе Ruby в качестве языка программирования для решения какой-либо задачи. Но после того как этот выбор сделан, они уже не конструктивны. Утиная типизация — это одна из возможностей языка и отказ от ее использования по каким-то философски-религиозным соображениям всего лишь означает уменьшение количества доступных разработчику инструментов. Практически любая возможность Ruby может быть использована как во вред, так и во благо, и утиная типизация здесь не исключение.

5 Пример с использованием OpenStruct и OptionParser

Отличным примером, демонстрирующим наиболее востребованные из описанных выше граней языка Ruby, является разбор аргументов командной строки с помощью стандартных классов OpenStruct [37] и OptionParser [29].

Приведенные ниже фрагменты взяты из реального, работающего Ruby-проекта. Этот проект состоит из нескольких автономных приложений командной строки, выполняющих различные части общей задачи по импорту в БД разнородной информации и ее последующей обработке. Функциональность приложений имеют достаточно много пересечений, поэтому общие куски функциональности вынесены в модуль Common.

Чтобы каждому приложению не приходилось реализовывать операцию разбора аргументов командной строки, реализован метод Common::parse_opt. Приложению нужно только вызвать его и передать ему набор обработчиков тех параметров, которые необходимы приложению:

```
class DataSchemaImportApp
  def run(args)
    options = Common::parse_opt(
        # Параметры командной строки.
        args,
        # Банер, который описывает назначение приложения и
        # способ его запуска.
        "Data Schema to DB importer\n" +
        "usage:\n" +
        "ruby dbsheme_import.rb <options>\n\n",
        # Обработчики отдельных опций.
        Common::DbOptionHandler.new,
        OptionHandler.new)
    ... # Дальнейшая обработка.
  end
  ...
end

# Запуск приложения на выполнение.
DataSchemaImportApp.new.run(ARGV)
```

Ниже приведена реализация метода Common::parse_opt. Использование различных особенностей языка Ruby описывается в комментариях:

```
# Список обработчиков параметров командной строки (параметр handlers)
# передается в виде аргумента Array, что позволяет разным приложениям
# использовать разное количество обработчиков.
#
def Common.parse_opt(args, banner, *handlers)
  opt = OptionParser.new

  # Здесь будут накапливаться уже разобранные параметры.
  # Объект OpenStruct не знает, какие именно значения будут храниться
  # в нем, поэтому он использует method_missing для сохранения новых
  # значений.
  res = OpenStruct.new(:verbose => false)

  # Банер потребуется объекту OptionParser когда придется выдавать
  # информацию о параметрах запуска приложения.
  opt.banner = banner

  # Объект opt типа OptionParser будет заниматься парсингом аргументов
  # командной строки. Поэтому всем обработчикам предоставляется
  # возможность описать в объекте opt нужные им параметры. Выполняется это в
  # итерации по всем элементам вектора handlers посредством метода each.
  # Здесь блок кода, в котором вызываются методы setup используется
  # для реализации внутреннего итератора.
  handlers.each do |h| h.setup(opt, res) if h.respond_to?(:setup) end

  # Аргументы --verbose и --help поддерживаются всеми приложениями.
  # Поэтому они описываются самим методом parse_opt.
  # Блоки кода здесь используются для обратного вызова -- OptionParser
  # вызовет их при обнаружении в командной строке соответствующего
  # аргумента.
  opt.on('-v', '--verbose', 'verbose output') do
    res.verbose = true
  end

  opt.on_tail("-h", "--help", "show this message") do
    puts opt
    exit
  end

  # Если аргументов командной строки нет, то объект OptionParser в своем
  # методе to_s сформирует описание приложения и его аргументов. Это
  # описание будет отображено на стандартный поток вывода, а работа
  # приложения -- завершена.
  begin puts opt; exit end if 0 == args.size

  begin
    opt.parse!(ARGV)

    # Еще одна итерация, в которой обработчикам дается возможность
    # проверить корректность разобранных параметров и их сочетаний.
    handlers.each do |h| h.check(res) if h.respond_to?(:check) end

  rescue StandardError => x
    $stderr.puts x.message
    exit
  end

  res
end
```

При работе с обработчиками параметров командной строки метод parse_opt придерживается утиной типизации — ему не важно, какому типу эти обработчики принадлежат и от кого они унаследованы. Ему даже не важно, есть ли у обработчиков оба метода setup (описание аргументов) и check (проверка аргументов) — вызываются только те, которые у объекта существуют. Такой подход дает свободу приложениям в реализации собственных обработчиков.

То, что parse_opt умеет работать с группами обработчиков параметров командной строки, а не с одним обработчиком, позволяет создавать отдельные классы обработчиков для каждой группы параметров. Например, для параметров, связанных с подключением к БД; для параметров, связанных с поиском файлов для импорта; для параметров статистической обработки и т.д. Соответственно, каждое приложение получает возможность использовать только те обработчики, которые необходимы именно ему.

Сам обработчик параметров может выглядеть, например, так:

```
class OptionHandler
  def setup(parser, result)
    result.unit_map = 'etc/unit_map.cfg'
    result.no_real_import = false

    parser.on('-U', '--unit-map CFG',
        'name of file with AAG units description') do |p|
      result.unit_map = p
    end

    parser.on('--no-real-import',
        'do not do real import into DB') do
      result.no_real_import = true
    end
  end
end
```

В методе setup производятся настройки не только объекта OptionParser, но и объекта OpenStruct, в котором собираются уже разобранные аргументы. Причем здесь блоки служат для реализации обратных вызовов, и используется тот факт, что блоки являются замыканиями — блоки остаются жить в программе и сохраняют связь с создавшим их контекстом даже после завершения работы setup.

6 Заключение

Вот таким получился небольшой рассказ о наиболее ярких, по моему мнению, гранях языка Ruby. Далеко не все возможности языка и его стандартной библиотеки вошли сюда, поскольку я старался писать только о том, что сам использовал при работе и что знаю достаточно хорошо. Не удалось также затронуть некоторые важные инструменты, необходимые при использовании Ruby, т.к. это могло увеличить рассказ вдвое. Может быть, получится сделать это в следующий раз.

В заключение же хочется сказать, что язык Ruby только на первый взгляд кажется простым. Действительно, у него очень низкий порог вхождения и решать простенькие задачки на Ruby можно уже через несколько часов после начала экспериментов с ним. Но по мере более плотного знакомства с языком приходит понимание, что Ruby многослоен и не так уж и прост. Кроме особенностей самого языка (например, таких как singleton-классы) существует еще один черезвычайно важный момент: в Ruby нет «единственно правильного» способа сделать что-либо. Поэтому Ruby-разработчик всегда имеет под рукой несколько средств для решения нужной ему задачи, и проблема выбора лучшего из этих средств лежит на его плечах. В этом смысле Ruby не проще, к примеру, C++. Может быть именно поэтому мне и интересен Ruby.

Благодарности

Автор выражает благодарность Леониду Борисенко, Зверьку Харьковскому и FR за помощь в подготовке статьи. Отдельное спасибо Игорю Мирончику за неоценимую помощь в корректуре и вычитывании текста статьи.

Список литературы

1. Dave Thomas, Andy Hunt, "Programming Ruby: The Pragmatic Programmers' Guide", Addison Wesley Longman, 2000.

Доступна в электронном виде: http://www.ruby-doc.org/docs/ProgrammingRuby/

1. Dave Thomas, Chad Fowler, Andy Hunt, "Programming Ruby: The Pragmatic Programmers' Guide. 2nd edition", 2005.
2. Hal Fulton. "The Ruby Way", Sams Publishing, 2001.
3. Why's (Poignant) Guide to Ruby, http://poignantguide.net/ruby/index.html. *Имеется русский перевод:* http://ruby.dmitriid.com/wiki/index.php
4. http://www.rubyonrails.org
5. http://www.ruby-lang.org
6. http://rubyforge.org/projects/rubyinstaller
7. http://rubyforge.org/projects/freeride
8. http://www.rubygems.org
9. http://www.mondrian-ide.com
10. http://rubyeclipse.sourceforge.net
11. http://www.activestate.com/Products/Komodo
12. http://www.ruby-ide.com/ruby/ruby_ide_and_ruby_editor.php
13. http://www.rubyforge.org
14. http://www.sourceforge.net
15. http://www.ruby-doc.org/docs/ruby-doc-bundle/Manual/man-1.4/index.html
16. Lucas Carlson, Leonard Richardson, "Ruby Cookbook", O'Reilly, 2006.
17. Robert Feldt, Lyle Johnson, "Ruby Developer's Guide", Syngress Publishing, 2002.
18. Dave Thomas, David Heinemier Hansson, "Agile Web Development with Rails", The Pragmatic Programmers LLC, 2005.
19. David A.Black, "Ruby for Rails: Ruby Techniques for Rails Developers", Manning Publications, 2006.
20. http://ru.wikibooks.org/wiki/Ruby
21. http://www.ruby-doc.org
22. http://www.gemjack.com
23. http://blade.nagaokaut.ac.jp/ruby/ruby-talk/index.shtml
24. http://www.martinfowler.com/bliki/DomainSpecificLanguage.html
25. http://martinfowler.com/articles/languageWorkbench.html. *Имеется русский перевод:* http://www.maxkir.com/sd/languageWorkbenches.html
26. http://rubyforge.org/projects/rake
27. http://eigenclass.org/hiki.rb?Changes+in+Ruby+1.9
28. http://www.ruby-doc.org/stdlib/libdoc/optparse/rdoc/classes/OptionParser.html
29. http://www.rubyforge.org/projects/fxruby
30. http://www.ruby-doc.org/stdlib/libdoc/tk/rdoc/index.html
31. http://whytheluckystiff.net/articles/seeingMetaclassesClearly.html
32. http://www.yaml.org
33. http://www.ruby-doc.org/core/classes/IO.html
34. http://www.ruby-doc.org/core/classes/Comparable.html
35. http://www.ruby-doc.org/core/classes/Enumerable.html
36. http://www.ruby-doc.org/stdlib/libdoc/optparse/rdoc/classes/OpenStruct.html
37. http://builder.rubyforge.org/