Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Московский государственный технический университет имени
Н.Э. Баумана
(национальный исследовательский университет)»

Р. С. Самарев

Введение в язык программирования Ruby

Учебное пособие

2-е издание, исправленное и дополненное

УДК 681.3.06 (075.8)
ББК 22.18
С17

Издание доступно в электронном виде на портале bmstu.press
по адресу: https://bmstu.press/catalog/item/7250/

Факультет «Информатика и системы управления»
Кафедра «Компьютерные системы и сети»

Рекомендовано Редакционно-издательским советом
МГТУ им. Н.Э. Баумана в качестве учебного пособия

Рецензент д-р хим. наук П.В. Слитиков

С17

Самарев, Р. С.

Основы языка программирования Ruby : учебное пособие / Самарев Р. С. ; МГТУ им. Н. Э. Баумана (национальный исследовательский ун-т). - 2-е изд., испр. и доп. - М. : Изд-во МГТУ им. Н. Э. Баумана, 2021. - 107 с. : рис., табл. - Библиогр.: С. 99. - ISBN 978-5-7038-5676-5.

Приведены сведения, необходимые для понимания языка программирования Ruby. Представлены примеры, рассматривающие ключевые для этого языка моменты. Последовательно раскрыты как базовые средства языка, так и его возможности, позволяющие существенно увеличить эффективность написания программ, включая объектные и функциональные принципы программирования.

Для студентов МГТУ имени Н.Э. Баумана, обучающихся по направлению «Информатика и вычислительная техника» и изучающих дисциплины «Языки интернет-программирования»

УДК 681.3.06 (075.8)
ББК 22.18

ISBN 978-5-7038-5676-5

Предисловие

Подготовка специалистов, имеющих отношение к вычислительным системам, подразумевает изучение современных технологий, получение практических навыков в использовании одного или нескольких языков программирования, а также понимание основных принципов построения языков программирования. При этом предполагается изучение как базовых алгоритмов, так и перспективных направлений развития программной отрасли. Невозможно предложить один-единственный язык программирования, который удовлетворил бы всем современным требованиям, поскольку большинство языков программирования имеет ту или иную специализацию.

Данное учебное пособие посвящено одному из популярных языков программирования — языку Ruby. Этот язык отличается стройной объектной моделью и чрезвычайной гибкостью, позволяющей быстро и эффективно реализовывать написанные на Ruby программы, начиная от вспомогательных программ и программ для автоматического тестирования приложения и заканчивая сложными веб-приложениями.

Язык Ruby для студентов полезен тем, что позволяет познакомиться с семейством скриптовых языков программирования, а также получить представления о функциональном программировании. При этом он может быть использован в очень широком диапазоне задач как учебного, так и практического характера. Для более глубокого понимания программирования на языке Ruby следует изучить работы [1–6].

Вопросы и замечания по данной работе просьба направлять автору на адрес: samarev@acm.org.

Введение

Язык Ruby разработан под влиянием таких языков, как Perl, Python, Smalltalk. Основной акцент сделан на удобство написания программы и последующего ее восприятия. Язык является мультипарадигменным. Условно, программист может писать программы в стиле языков типа Pascal, C/C++ или в стиле скриптовых языков программирования типа Perl, Python. Это полезно для тех, кто только начал осваивать Ruby.

Ruby обладает развитой объектной моделью, заимствованной от Smalltalk, имеет изначально заложенные средства, дающие возможность писать программы в стиле функциональных языков программирования.

Язык Ruby использует кодировку UNICODE, поэтому программа может быть написана даже на русском языке, включая названия классов, методов, переменных.

Основные принципы Ruby – минимальное количество кода при максимальной его выразительности, а код программы должен восприниматься как код на естественном языке. Кажущаяся избыточность способов реализации одной и той же программы (за что Ruby критикуют те, кто его не использует) обусловлена предоставлением программисту возможности выбрать именно те имена методов, которые являются наиболее выразительными в данном конкретном случае с позиции человеческого языка общения.

1. БАЗОВЫЙ СИНТАКСИС

Язык Ruby является объектным языком программирования с динамической типизацией и автоматической очисткой памяти. Это означает, что объект может иметь тип, но переменные или константы, которые на него ссылаются, не имеют сами по себе никакого типа. Объекты, на которые нет ссылок, будут автоматически удалены.

Для написания программы на языке Ruby до версии 2.0 использовалась 7-битная ASCII-кодировка. В Ruby версии 2.0 по умолчанию применяется кодировка UTF-8. Идентификаторы можно написать на русском языке. Комментарии и строки могут содержать символы в любой кодировке, однако это необходимо указывать явно.

В строке может быть сколь угодно много лексем, разделенных пробелами. Если в одной строке пишется несколько выражений, необходимо использовать точку с запятой, иначе строка может быть отброшена.

В современных версиях Ruby логические выражения и последовательности вызова методов можно переносить на несколько строк. В ранних версиях Ruby (до версии 1.9) для этого необходимо было использовать символ переноса строки — обратную косую черту «\».

Программа выполняется сверху вниз. Отдельной главной функции не существует, однако возможно организовать проверку какой именно файл запущен на выполнение. Если по тексту встречается код функций (методов), он будет выполнен только в том случае, если был осуществлен их явный вызов.

Зарезервированные слова (для справки):

```
alias     and     BEGIN   begin    break   case    class    def

defined?  do      else    elsif    END     end     ensure   false
for       if      in      module   next    nil     not      or
redo      rescue  retry   return   self    super   then     true
undef     unless  until   when     while   yield
```

Скобки при вызове методов можно опускать, например:

```
foobar
foobar()
foobar a, b, c
foobar( a, b, c )
```

До версии языка Ruby 2.0 были существенны позиции пробелов.

Следующие выражения эквивалентны:

```
x = y + z
x = y+z
x = y+ z
```

Однако выражение

```
x = y +z
```

не эквивалентно выражениям, приведенным выше.

Последнее выражение интерпретировалось как вызов метода «y» с параметром +z, т. е. x = y(+z). В современных версиях языка эта операция рассматривается как сложение. Также может быть неоднозначность с разбором выражений тип:

```
foobar(a, b)
foobar  (a), (b)
foobar  (a, b)  # syntax error, unexpected ',', expecting ')'
```

Особенность последней строки в том, что пробел после имени метода заставляет Ruby воспринимать остальную часть строки как отдельные аргументы. То есть, ближайшая скобка интерпретируется как составной оператор. Аналогичная ситуация в интерпретации пробелов у операции взятия элемента массива по индексу: x[1] без пробела против x [1], то есть x([1]).

Ключевое слово then для if не является обязательным. Блок (или код, который должен быть выполнен для указанного метода) может быть представлен в коде программы либо символом «{…}», либо словами do … end.

Основной принцип Ruby — DRY (Don't repeat yourself) code, что означает в вольном переводе «сухой код без повторений». Кроме того, в концепции Ruby делается акцент на то, что программа должна быть написана на языке, максимально приближенном к естественному (например, к английскому).

Студентам рекомендуется самостоятельно проверить все приведенные в пособии примеры. Для этого необходимо знать, что программа на языке Ruby — это текстовый файл, обычно имеющий расширение rb, который может быть запущен в консоли командой: ruby file.rb. Для отладки программ или изучения Ruby полезно использовать команду irb (т. е. интерактивный Ruby), которая позволяет вводить или выполнять операции построчно. Процесс установки Ruby описан в приложении 1.

1.1. Правила именования

В языке Ruby существуют довольно простые правила именования переменных, методов и классов. Их знание необходимо как для понимания смысла ранее созданных программ, так и для написания программ, которые будут понятны другим. Написание идентификатора определяет его предназначение (empty**?** или empty**!**).

Имена идентификаторов приведены в табл. 1.1.

Таблица 1.1

Имена идентификаторов

Запись идентификатора	Назначение	Пример
Со строчной буквы или со знака '_'	Имена локальных переменных	local_variable __FILE__
Со знака доллара **$**	Имена глобальных переменных	**$**global_variable
Со знака **@**	Переменные экземпляра класса	**@**instance_variable
Со знака **@@**	Переменные класса	**@@**class_variable
С прописной буквы	Имена констант	TRUE

Правила именования применительно к методам классов заключаются в использовании суффиксов метода (#empty**?**, #sub**!**,...):

если «**?**» — метод является предикатом, т. е. утверждением с результатом ИСТИНА или ЛОЖЬ;

если «!» — метод производит изменение данных, т. е. является деструктивным.

Примеры использования методов классов с указанными выше суффиксами:

```
obj.empty? # объект пуст? (истина/ложь)

Numeric.nonzero? # число не нуль? (истина/ложь)

obj.upcase! # Перевести буквы в верхний регистр

obj.ident! # Добавить отступы в строку
obj.capitalize! # Сделать все слова с заглавной буквы
```

Обратите внимание на то, что метод obj.ident без суффикса **!** не должен изменять объект obj, а должен возвращать измененную копию модифицируемых данных!

1.2. Предопределенные переменные и константы

В качестве справочной информации приведем списки псевдопеременных и предопределенных переменных Ruby, которые доступны в любом месте программы. Отметим, что бóльшая часть специальных переменных унаследована из языка Perl и редко используется при программировании на Ruby.

Псевдопеременные представлены в табл. 1.2, предопределенные переменные — в табл. 1.3, глобальные константы — в табл. 1.4.

Таблица 1.2

Псевдопеременные

Имя переменной	Описание
Self	Объект, выполняющий данный метод
Nil	Единственный экземпляр класса NilClass (представляет false)
True	Единственный экземпляр класса TrueClass (ИСТИНА)
False	Единственный экземпляр класса FalseClass (ЛОЖЬ)
__FILE__	Имя текущего файла
__LINE__	Номер строки выполнения в текущем файле

Таблица 1.3

Предопределенные переменные

Имя переменной	Описание
$!	Текст сообщения об исключении, установленном инструкцией 'raise'
$@	Массив стека вызовов на момент последнего исключения
$&	Строка, соответствующая последнему успешному совпадению при поиске по шаблону
$`	Содержит строку слева от шаблона последнего успешного поиска
$'	Содержит строку справа от шаблона последнего успешного поиска
$+	Содержит последнюю группу символов, соответствующую шаблону последнего успешного поиска
$1	Содержит N-ю группу символов, соответствующую шаблону последнего успешного поиска. Может быть больше единицы
$~	Информация о шаблоне последнего поиска
$=	Флаг нечувствительного к регистру поиска. По умолчанию nil
$/	Разделитель записей. По умолчанию перевод строки
$\	Разделитель записей при выводе для print и IO#write. По умолчанию nil
$,	Разделитель полей при выводе для print и Array#join
$;	Разделитель по умолчанию для String#split
$.	Номер строки, прочитанной из файла
$>	Вывод по умолчанию для print, printf. По умолчанию $stdout
$_	Последняя полученная методами gets или readline строка
$0	Содержит имя выполняемого в данный момент скрипта. Может быть переопределен
$*	Параметры командной строки, переданные скрипту

Окончание табл. 1.3

Имя переменной	Описание
$$	Идентификатор процесса Ruby, который выполняет данный скрипт
$?	Статус выполнении последнего запущенного дочернего процесса
$:	Путь загрузки для скриптов и бинарных модулей для инструкций «load» или «require»
$"	Массив, который содержит имена модулей, загруженных инструкцией «require»
$DEBUG	Состояние опции -d
$FILENAME	Имя файла ввода из переменной $<. Аналогично вызову $<.filename
$LOAD_PATH	Синоним $:
$stderr	Стандартный вывод ошибок
$stdin	Стандартный ввод
$stdout	Стандартный вывод
$VERBOSE	Флаг подробного вывода сообщений, который устанавливается опцией ruby -v
$-0	Синоним $/
$-a	True, если опция -a установлена. Только чтение
$-d	Синоним $DEBUG.
$-F	Синоним $;
$-I	Синоним $:
$-l	True, если опция -l установлена. Только чтение
$-p	True, если опция -p установлена. Только чтение
$-v	Синоним $VERBOSE
$-w	True, если опция -w установлена

Таблица 1.4

Предопределенные глобальные константы

Имя переменной	Описание
TRUE	true
FALSE	false
NIL	nil
STDIN	Стандартный ввод. По умолчанию $stdin
STDOUT	Стандартный вывод. По умолчанию $stdout
STDERR	Стандартный вывод ошибок. По умолчанию $stderr
ENV	Хеш (коллекция пар ключ — значение), который содержит переменные окружения
ARGF	Синоним $<
ARGV	Синоним $*
DATA	Бинарные данные за директивой __END__ в коде, если она присутствует. Иначе не определена
RUBY_VERSION	Строка версии Ruby (константа VERSION устарела)
RUBY_RELEASE_DATE	Строка даты выпуска Ruby
RUBY_PLATFORM	Идентификатор платформы

1.3. Комментарии

Однострочные комментарии устанавливаются символом «#» и распространяется он до конца текущей строки:

```
# Выражение на Ruby

if b == 2
 true      # случай 1
else
 prime?(b) # случай 2
end
```

Многострочные комментарии:

```
=begin

Пример многострочного комментария
print "Какой-то текст для отображения".
=end
```

Обратите внимание на то, что перед **=begin** и **=end** не должно быть абсолютно никаких символов, включая пробелы!

Существуют также специальные комментарии, которые могут быть вставлены в начале программы, например комментарий, указывающий кодировку текста программы,

```
# coding: utf-8
```

1.4. Константы, переменные

Ранее были рассмотрены основные принципы именования констант и переменных. Добавим, что предварительная декларация до их использования не требуется.

Для Ruby не существует атомарных типов данных. Любые данные являются объектами тех или иных классов. Операции над объектами порождают новые объекты за исключением некоторых специальных случаев (методов, содержащих в имени суффикс '!'). И константы, и переменные хранят ссылку на объект. Их существование начинается в момент первого присвоения этой ссылки (включая nil).

Константа не может быть инициализирована дважды. Например, выполнение кода

```
A='s1'; A='s2'
```

выдаст предупреждение

```
'warning: already initialized constant A'.
```

Любой переменной может быть присвоена ссылка на любой объект. Отметим, что Ruby является языком с динамическим контролем типов, поэтому проверка осуществляется в момент выполнения операций, но не в момент присвоения переменной ссылки на объект. Например, корректен следующий код, поскольку переменной ***a*** последовательно присваиваются ссылки на объекты разных типов:

```
a='s1'; a = 1; a = a+1
```

Но не корректен код

```
a='s1'; a = a+1
```

для которого имеем ошибку несоответствия типов объектов при сложении:

```
”TypeError: can't convert Fixnum into String”.
```

Отметим, что следующая операция корректна и приведет к склейке строк и созданию объекта 's11', поскольку для класса String существует реализация метода «+» для конкатенации:

```
a='s1'; a = a+'1'
```

Поясним более подробно особенность работы с переменными и константами. См. рис. 1.1.

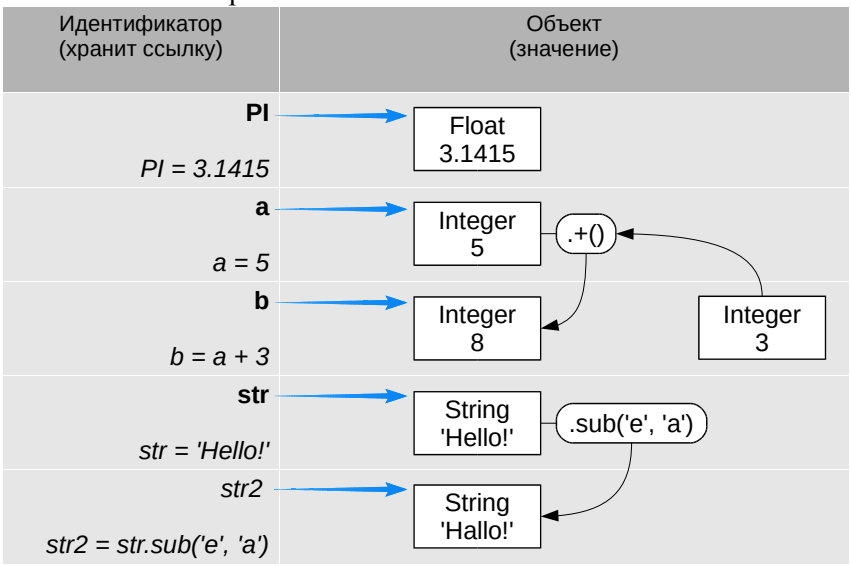

Рис. 1.1. Переменные и константы

Константа PI здесь ссылается на объект типа Float, имеющий значение 3.1415. Переменная **a** ссылается на объект типа Integer со значением 5. Переменная **b** получает результат операции a+3, которая является методом #+ объекта 5, а в качестве аргумента выступает объект типа Integer, имеющий значение 3. Результатом выполнения этого метода (сложения), является создание нового

объекта типа Integer со значением 8, ссылку на который и получит переменная b.

В следующем примере, str получает ссылку на объект типа String со значением „Hello“. А переменная str2 получает ссылку на объект, порождённый методом sub (substitute — замена).

Обратите внимание, что здесь приводятся пояснения в терминах модели языка Ruby, но не с позиции внутренней реализации механизмов работы с числами. Но, тем не менее, можно видеть, что для Ruby есть сущности переменная и константа с одной стороны, и объекты с другой стороны. Причем вызов какого-либо метода объекта, как правило, приводит к созданию нового объекта (если имя метода не содержит суффикс — восклицательный знак). Эта схема справедлива как для простых арифметических выражений, так и для имен классов, модулей, методов и пр.

1.5. Область видимости переменных и констант

Локальные переменные доступны после момента инициализации во всех вложенных по отношению к ним блоках (переменные, которые декларируются между символами «|...|» для блоков, ограниченных служебными словами do...end и символами «{...}»).

Пример:

```
str = '1'

5.times do |i|
        i.times{ printf str }
        puts
end
```

То же правило действует для анонимных процедур Proc, причем такой код будет получать последнее значение переменной, например:

```
str = 'test'

func = Proc.new {puts str}
func.call   # => test
str = 123
func.call   # => 123
```

Обратите внимание, что внутрь методов вложенность не распространяется. Следующий пример вызовет ошибку:

```
str1 = 'test 1'

puts str1
# декларируем метод
def print_str1
  puts str1 # => undefined local variable or method `str1'
end
# вызываем метод
print_str1
```

Глобальные переменные (имеют префикс **$**) доступны везде, но после того, как они были инициализированы:

```
# декларируем метод

def print_globvar
  puts $str # печатаем создаем глобальную, если она существует
end
if true
   #создаем и инициализируем глобальную переменную
   $str = '1'
end
puts $str     # печатаем создаем глобальную
print_globvar # вызываем метод, в котором есть обращение
              # к переменной
```

Константы, идентификатор которых начинается с прописной буквы, по области видимости аналогичны глобальным переменным:

```
if true

  STR = '1'    #создаем константу
end

puts STR    #печатаем константу

# декларируем метод
def print_globvar
  puts STR  #печатаем константу
end
print_globvar
```

1.6. Простейший консольный вывод

Рассмотрим методы вывода в консоль с тем, чтобы предоставить возможность читателю осознанно запускать

приведенные здесь примеры. Для вывода в консоль могут использоваться следующие методы:

• print obj — печать содержимого объекта. Если объектом является строка или число, осуществляется вывод без изменений; в противном случае выполняется неявное преобразование объекта в строку;

• puts obj — печать содержимого объекта с добавлением символа перевода строки;

• printf format_str, args — метод вывода по форматной строке, аналогичный C/C++;

• p obj — специальный метод для вывода отладочной информации о структуре объекта. Эквивалентен вызову puts obj.inspect. Формат выводимой информации для основных типов данных соответствует записи литералов для этих типов.

Контрольные вопросы и задания

1. Каково назначение следующих идентификаторов с точки зрения Ruby:

A, $a, _A, a, @a, @@a

2. Допустима ли следующая запись (на русском языке):

число_1 = 2; **число_2** = **число_1***5?

3. Допустимы ли следующие выражения:

a = 1; a = '123';?

4. Какие из приведенных ниже выражений дают разные результаты:

```
x = y  +  z
x =   y+z
x = y+    z
x = y    +z?
```

Для проверки запустите интерактивный ruby — irb, определите значения переменных x и y, поочередно введите каждую строку и сравните результаты.

5. Возможен ли следующий комментарий:

```
a = c  + # получено ранее #  b
```

6. В чем заключается принципиальное различие следующих методов (по именам):

```
obj.ident и obj.ident! ?
```

2. ОСНОВНЫЕ КОНСТРУКЦИИ ЯЗЫКА

2.1. Основные типы

Приведем небольшой список часто применяемых встроенных типов данных:

- Fixnum/Bignum (целые числа, меньше 2^{30} / больше 2^{30});
- Float (числа с плавающей запятой);
- Array (массивы);
- String (строки);
- Hash (ассоциативные массивы);
- Symbol (константная строка);
- Range (диапазон);
- Regexp (регулярное выражение).

Литералы указанных типов, чаще всего, используются без указания имени класса. Типы Fixnum/Bignum в современных версиях Ruby заменены Integer для всех случаев.

В полном списке присутствует порядка 200 классов, поэтому приводить его в пособии не будем. Краткий пример иерархии классов представлен на рис. 2.1.

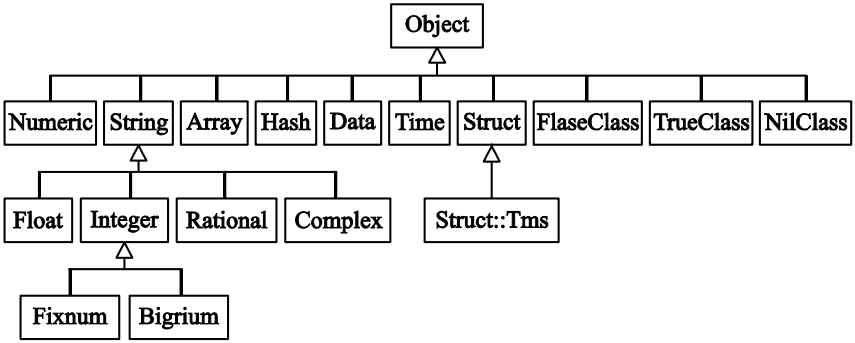

Рис. 2.1. Иерархия классов

Числовые литералы в Ruby выглядят следующим образом:

```
5     # целое число

-12   # отрицательное целое число
4.5   # число с плавающей запятой
076   # восьмеричное число
0b010 # двоичное число
0x89  # шестнадцатиричное число
```

Логический тип в Ruby представлен двумя предопределенными переменными true (ИСТИНА или ДА) и false (ЛОЖЬ или НЕТ). Логический тип появляется в результате логических операций или вызова логических методов.

Необходимо отметить следующее:

• традиционно имена логических методов заканчиваются на знак «?»;

• в качестве false может выступать nil, а в качестве true — любой объект;

• nil — символ пустоты или отсутствия объекта.

2.2. Операторы

Основные операторы Ruby представлены в табл. 2.1.

Таблица 2.1

Основные операторы Ruby по приоритету

Оператор	Описание
::	Разрешение области видимости
[]	Взятие индекса
**	Возведение в степень
+ - ! ~	Унарный плюс/минус, логическое отрицание, битная инверсия
* / %	Умножение, деление, остаток от деления
+ -	Сложение/вычитание
<< >>	Операторы поразрядного сдвига
&	Поразрядное И
| ^	Поразрядное ИЛИ, исключающее ИЛИ
> >= < <=	Сравнение

Окончание табл. 2.1

Оператор	Описание
== === <=> != =~ !~	Равенство, комбинированное сравнение (–1, 0, 1), неравенство, операторы сравнения по шаблону
&&	Логическое И
||	Логическое ИЛИ
.. ...	Операторы диапазона
= (+=, -=, …)	Присваивание
?:	Тернарный выбор
not	Логическое отрицание
and or	Логическое И, ИЛИ (отличаются от операций && и || низким приоритетом)

Существует операция множественного присваивания, впрочем, с точки зрения современного стиля программирования, она не рекомендуется к использованию, так как может привести к ошибкам восприятия:

```
a,b = c,d; a,b=[1,2]; a,b,c=c,a,b
```

Обратите внимание на то, что операции !, &&, || и not, and, or имеют различный приоритет по отношению к операциям сравнения и присвоения, например:

```
a = 'test'

b = nil
both = a && b        # both => nil
both = a and b       # both => 'test'
both = (a and b)     # both => nil
```

Имеется особенность в приведении значений к логическому типу. Рассмотрим пример программы, формирующей массив различных значений и проверяющей их на истинность:

```
[nil, 0, 1, true, false, '', '123'].each do |i|
       puts i.inspect + "\t is true" if i
end
```

Результат выполнения программы:

```
0       is true

1       is true
true    is true
""      is true
"123"   is true
```

Серьезное отличие Ruby от языков C, C++ заключается в том, что любое числовое значение есть true. **Запомните, что только false и nil есть ЛОЖЬ!** Все остальные объекты истинны (т. е. true), поскольку они существуют. Строго говоря, false и nil — также объекты, но для них сделано исключение.

2.3. Блоки

В языке Ruby в явном виде нет операторного блока, привычного для таких языков, как Pascal — конструкция do .. end или C — символ «{...}». Оба варианта в Ruby присутствуют, однако не могут существовать без указания метода, к которому они относятся. Например, автономное использование выражения {puts '*'} недопустимо, в то время как запись 10.times {puts '*'} корректна. Основной принцип выбора способа написания между словами do...end и символами «{}» заключается в том, что фигурные скобки следует применять в однострочных операторных блоках, поскольку в этом случае код получается более компактным и лучше читаемым. Если операторный блок многострочный, следует применять конструкцию do…end.

Ниже представлены два эквивалентных фрагмента кода:

```
a = (1..3)

puts a.map { |n|  n = Math.sin(n); n*2 }.to_s
puts( a.map do |n|
            n = Math.sin(n)
          (n*2)
      end.to_s )
```

Для методов puts, print блок do (см. http://www.skorks.com/2009/09/using-ruby-blocks-and-rolling-your-own-iterators/) имеет высший приоритет, поэтому следующие эквивалентные примеры не работают:

```
puts a.map do |n|
       n = Math.sin(n); (n*2)
end
puts (a.map) { |n|  n = Math.sin(n); n*2 }
```

2.4. Циклы и ветвление

Ruby является скриптовым языком, поэтому синтаксис его разработан так, чтобы с минимумом кода получить наиболее выразительный и значимый результат. Записать одно и то же можно различными способами, а какой способ применить в каждом случае решает программист в соответствии с контекстом.

Для написания условий применяются выражения и модификаторы.

Ниже приведены примеры выражений.

Традиционное «если»...«то»...«иначе» :

```
if conditional [then]

         code...
[elsif conditional [then]
         code...]...
[else
         code...]
end
```

Выражение «если не»...«то»...«иначе», которого нет в C/C++:

```
 unless conditional [then]

    code
 [else
    code ]
 end
```

Множественный выбор:

```
 case expression

 [when expression [, expression ...] [then]
    code ]...
 [else
    code ]
 end
```

Модификаторы используются для коротких однострочных записей и отличаются от выражений обратным порядком следования условия и выполняемого действия, например:

```
code if condition

code unless conditional
```

Подробнее ознакомиться с циклом и ветвлением можно в книге [1], однако ряд примеров из этой книги приведем здесь. Примеры написания условий представлены в табл. 2.2.

Таблица 2.2

Примеры написания условий

Форма с if	Форма с unless

`if x < 5 then`
`  выражение`
`end`
	`unless x >= 5 then`
	`  выражение`
	`end`

`if x > 2`
 	
`  puts "x больше, чем 2"`
`elsif x <= 2 and x!=0`
`  puts "x равно 1"`
`else`
`  puts "Не могу отгадать номер"`
`end`
	`unless x > 2 then`
	
	`  puts "x меньше, чем 2"`
	`else`
	`  puts "x больше, чем 2"`
	`end`

`print "Значение определено\n"` `if $var`	`print "Значение не определено\` `n" unless $var`
`x = if a>0 then b else c end`	`x = unless a<=0 then c else b` `end`

В отношении циклов, Ruby предоставляет множество способов их организации. Например следующие циклы весьма традиционных для всех языков программирования:

```
# Цикл for для массива
for x in list do
  print "#{x} "
end

# Цикл for для диапазона
n = list.size - 1
for i in 0..n do
  print "#{list[i]} "
end

# Цикл 'loop'
i = 0;       n = list.size - 1
loop do
  print "#{list[i]} "
  i += 1
  break if i > n
  # break unless i <= n
end
```

```
# Цикл while
i = 0
while i < list.size do
  print "#{list[i]} "
  i += 1
end

# Цикл until
i = 0
until i == list.size do
  print "#{list[i]} "
  i += 1
end
```

Однако, вспоминая, что Ruby разработан как язык, выражающий смысл для человека-носителя естественного языка, то глядя на эти циклы, делаем вывод лишь о том, что понятны они только программисту, но не носителю языка. Именно поэтому подобные циклы почти не используются в Ruby-программах. Вместо них должны использоваться специальные методы, прямо выражающие смысл действия. Примеры:

```
# Цикл итератор 'upto' – от меньшего до большего
n = list.size - 1
0.upto(n) { |i| print i, ' ' }
#=> 0 1 2 3 4  ...

# Цикл итератор 'downto' – от большего до меньшего
5.downto(1) { |n| print n, '.. ' }
#=> "5.. 4.. 3.. 2.. 1..

# Цикл 'times' – … раз повторить
n = list.size
5.times {|i| print i, ' ' }
#=> 0 1 2 3 4

# Цикл итератор 'each' – для каждого элемента
list.each do |x|
  print "#{x} "
end

# Цикл итератор индекса 'each_index' – для каждого индекса
list.each_index do |i|
  print "#{list[i]} "
end

# Посчитать количество 'count'
puts (0..n).count { |i| i.odd? }

# Найти по условию 'detect'
puts [0, 1, 2, 3].detect { |i| i % 5 == 0 }
puts [0, 1, 2, 3].find { |i| i % 5 == 0 }
….
```

Приведенные здесь примеры методов — лишь малая часть доступных методов. Полный их перечень см. в документации для модуля Enumerable - https://ruby-doc.org/core-2.7.0/Enumerable.html

Подводя небольшой итог, отметим, что для языка Ruby следует максимально использовать выражения, позволяющие коротко выразить что именно происходит в этом конкретном месте кода. Например счётный цикл for можно заменить вызовом метода #times (т. е. «сколько раз повторить») или методом #each_with_index (т. е. «для каждого элемента вместе с индексом») в тех случаях, когда нужен и элемент, и его индекс. Естественно, внутри все эти методы могут использовать традиционные циклы, но снаружи они используются именно как осмысленные действия, в не абстрактные for, while, loop.

2.5. Исключения

Исключения являются механизмом для обработки ошибок выполнения кода.

Генерация исключения реализуется вызовом метода raise:

```
raise ExceptionClass[, "message"]
```

Обработка исключений реализуется следующим образом:

```
begin

  # защищаемый блок кода
  expr..
[rescue [error_type [=> var],..]
  # исключение с типом error_type
  expr..]..
[else
  # по всем остальным ошибкам сюда
  expr..]
[ensure
  # по окончании вызвать этот код независимо того,
  # были ошибки или нет
  expr..]
end
```

Поскольку выбрасывание исключения обычно является чем-то неординарным, перехватывать следует только исключения того типа, с которым точно известно, что следует делать. В противном случае исключение должно быть передано выше по стеку (это

произойдет автоматически, если в rescue не будет найден соответствующий тип).

2.6. Основы классов

В Ruby все есть объект. Даже имя класса объекта — это экземпляр системного класса Class. А код метода или блока является объектом класса Proc. Класс Object является суперклассом.

Определение класса объекта:

```
class Identifier [< superclass ]
  expr..
end
```

Обратите внимание на то, что имя класса всегда начинается с заглавной буквы. Это индикатор того, что имя является константой.

Константа указывает на объект, класс которого можно определить, «вызвав метод» class. Например, для стандартного класса String получим

```
puts String.class # => Class
```

Синглтон, т. е. одиночный глобальный экземпляр класса:

```
obj_name=[]

class << obj_name
  expr..
end
```

Пример использования синглтона:

```
log=[]

class << log
  def out() puts "Hi!" end
end

log.out
```

Определение переменных и методов рассмотрим на следующем примере. Определим класс MyTest с переменной @@title для всех экземпляров данного класса переменными @name, @result для каждого экземпляра в отдельности и методами print_result и print_title.

```
class MyTest

  @@title = "Результаты измерений" #переменная  класса
  def initialize(name, result)
    @name, @result = name, result  # переменные экземпляра
  end

  def print_result
    puts "#{@name}: #{@result}"
  end

  def self.print_title
    puts @@title
  end
end

t1 = MyTest.new("измерение 1",10)
t2 = MyTest.new("измерение 2",50)

MyTest.print_title     # Результаты измерений
t1.print_result               # измерение 1: 10
t2.print_result               # измерение 2: 50
```

Поскольку Ruby имеет средства, позволяющие прочитать из собственной программы действующую иерархию классов, приведем программу для формирования SVG-файла, содержащего всю иерархию классов Ruby (без дополнительных модулей):

```
anc_desc = {}

ObjectSpace.each_object(Class).
   select {|x| x < Object}.each
   {|c| anc_desc[c.name]=c.superclass.name}
File.open 'result.dot', 'w' do |file|
  file.puts %Q(digraph "Ruby #{RUBY_VERSION}" {\n)
  file.puts %Q(node [shape=box];\n edge [arrowtail="empty",
dir=back];\n)
  anc_desc.each {|desc, anc|  file.puts %Q("#{anc}" -> "#{desc}";\
n)}
  file.puts '}';
end
system "dot -Tsvg result.dot -o ruby.svg"
```

Для корректного ее функционирования необходимо, чтобы было установлено средство graphviz, в состав которого входит утилита **dot**.

2.7. Строки

Строки Ruby реализуются классом String. Создание строки может осуществляться через присвоение литерала либо с использованием явного вызова конструктора String.new. В версиях Ruby до 1.8 символы строки были 8-битными. Начиная с версии 1.9, внутренним представлением символов является UNICODE. (Дополнительную информацию о строках можно получить из документации https://ruby-doc.org/core-2.7.0/String.html)

Строковые литералы в Ruby могут быть с одинарными или двойными кавычками.

При анализе строки в одинарных кавычках распознаются только две специальные последовательности «\\» и «\'»:

```
str = 'Строка'
path = 'c:\\windows\\'
str2 = 'Переменная \'str\' содержит строку'
```

Строка в двойных кавычках позволяет включать специальные управляющие символы:

```
str = "Знак табуляции: \t"
str = "Перенос \n строки"
str = "Еще один знак табуляции \011"
```

Существует альтернативная нотация строк с использованием записи %q[…] или %Q[…], которые соответствуют строкам с одиночными и двойными кавычками соответственно, причем скобки могут быть любыми. Эта форма записи строк удобна тем, что внутри скобок можно применять кавычки без дополнительного экранирования, что актуально при работе со строками, содержащими разметку XML или HTML:

```
str = %q[Строка с символом переноса \n, но отображаемая как
написано]

str = %Q[Строка с переносом \n строки]
```

Еще один встроенный способ включения многострочных документов непосредственно в код заключается в использовании служебного слова EOF:

```
str = <<EOF
Некоторый
   многострочный
      текст
         с отступами, которые так и перейдут в строку.
```

```
EOF
```

Поскольку все, что осуществляется над строками, явно или неявно реализуется методами класса String, рассмотрим его методы и примеси (один из способов расширения функциональности класса).

2.7.1. Конструирование объектов класса String

Класс String содержит метод new, который создает копию строки, переданной в качестве аргумента.

Пример:

```
# coding: utf-8

str1 = "Некоторая строка"
str2 = str1
str3 = String.new(str1)

#заменим гласные в первой строке на *
str1.gsub!(/[еоая]/, '*') # использован gsub!, а не gsub

puts str1 # Н*к*т*р** стр*к*
puts str2 # Н*к*т*р** стр*к*
puts str3 # Некоторая строка
```

Обратите внимание на то, что str1 и str2 выдали одну и ту же строку, хотя модификация была произведена над str1. Это говорит о том, что переменная str2 указывала на тот же объект, что и str1, а переменная str3 содержала копию с объекта, на который изначально указывала str1, которую создал оператор String.new.

Метод gsub обеспечивает замену по регулярному выражению во всем тексте сразу. Обратите внимание, что использована форма gsub!, а не gsub. Это значит, что модифицирован сам объект, который вызвал этот метод. Метод gsub создал бы измененную копию объекта.

Следует отметить, что копирование строки можно реализовать как через создание нового объекта,

```
str3 = String.new(str1)
```

так и с помощью операции клонирования

```
str3 = str1.clone
```

2.7.2. Методы класса String

В отличие от метода new остальные методы могут быть применены только к конкретным объектам, а не классу в целом. Назначение большинства методов понятно из их названия. Для подробного их изучения следует обратиться к документации на класс String – https://ruby-doc.org/core-2.5.0/String.html .

Таблица 2.3

Некоторые методы для работы со строками

Имя метода	Назначение

`#[]=, #[]`	Записать и прочитать по индексу из строки:
	`a = "hello there"`
	`a[1]  #=> "e"`
	`a[2] = 'm'`
	В качестве индекса могут также использоваться подстроки и регулярные выражения

`#%`	Форматирование строки:
	`"%05d" % 123  #=> "00123"`
	
	Значения для подстановки заданы массивом и соответствуют позициям элементов:
	`"%-5s: %08x" % [ "ID", self.object_id ]`
	`#=> "ID  : 200e14d6"`
	Значения для подстановки заданы ассоциативным массивом и доступны по имени:
	`"foo = %{foo}" % { :foo => 'bar' }`
	`#=> "foo = bar"`

`#*`	Размножение строк
	`"Ho! " * 3  #=> "Ho! Ho! Ho! "`
	`"Ho! " * 0  #=> ""`

`#insert`	Вставить подстроку в указанную позицию

`#<<, #concat,`	Добавить укакзанную строку в конец текущей
	`a = "hello "`
	`a << "world"  #=> "hello world"`
	`a.concat(33)  #=> "hello world!"`
	`a  #=> "hollo world!"`

`#+`	Склеить строки и создать новый объект

Имя метода	Назначение

`#<=>,`	Сравнить на совпадение (0), меньше (-1), больше (1). Сравнение лексикографическое.
	`"abcdef" <=> "abcde"  #=> 1`
	`"abcdef" <=> "abcdef"  #=> 0`
	`"abcdef" <=> "abcdefg"  #=> -1`
	`"abcdef" <=> "ABCDEF"  #=> 1`
	`"abcdef" <=> 1  #=> nil`

`#==, #eql?`	Сравнить строки. true если совпали.
`#casecmp,` `#casecmp?`	Аналогичные методы сравнения строк **без учета регистра**. casecmp соответствует <=>. casecmp? соответствует eql?

`#capitalize!,`
`#capitalize`
	Сделать первую букву заглавной, остальные - строчные. Модифицирует текущую строку (метод с восклицательным знаком) или возвращает новую модифицированную.
	`"HELLO".capitalize`
	`#-> Новая строка "Hello"`
	`a = "hello";`
	`a.capitalize!  #изменение a! "Hello"`

`#center,`
`#ljust,`
`#rjust,`
	Выровнять текст по центру / слева / справа, заполнив до указанной ширины по краям пробелами или указанными подстроками.
	`"hello".center(20)`
	`#=> "       hello        "`
	`"hello".ljust(20)`
	`#=> "hello               "`
	`"hello".rjust(20)`
	`#=> "               hello"`

`#chomp!, #chomp`	Удалить символ-разделитель с конца строки, если есть. Обычно используется для удаления символа перевода строки. `"hello\r\n".chomp  #=> "hello"`
`#chop!, #chop`	Безусловно удалить последний символ строки.

`#count`	Подсчитать количество входждений заданных последовательностей символов или подстрок.
	`a = "hello world"`
	`a.count "lo"  #=> 5`
	`a.count "lo", "o"  #=> 2`

Имя метода	Назначение
`#crypt, #hash`	Вычислить криптографический или простой хэш по указанной строке
`#delete!, #delete`	Удалить из строки укакзанные последовательности или символы из заданного набора `"hello".delete "l","lo"` **`#=> "heo"`** `"hello".delete "lo"` **`#=> "he"`**

`#downcase!,`
`#downcase,`
`#upcase!,`
`#upcase,`
`#swapcase!,`
`#swapcase`
	Перевести все символы строки в нижний, верхний регистр или поменять их наоборот. Имеются ограничения на кодировку символов, для которой эти методы будут работать. Поддерживаются UTF-8, UTF-16BE/LE, UTF-32BE/LE, and ISO-8859-1~16 Strings/Symbols.

`#dump`	Сформировать новую строку, преобразовав не печатные символы в их коды. `"hello \n ''".dump` **`#=> "\"hello \\n ''\""`**

`#each_byte,`
`#each_char,`
`#each_line,`
`#each`
	Получить объект Enumerator или обработать в блоке каждый байт, символ, линию

`#sub!, #sub,` `#gsub!, #gsub,`	Выполнить замену подстроки в строке. Sub – только первое вхождение, gsub – global substitute – заменить все вхождения.
`#replace`	Заменить всю строку на новую заданную в текущем объекте
`#include?`	Определить входит ли подстрока в строку
`#index, #rindex`	Найти позицию вхождения подстроки в строку с начала или с конца строки. Иначе – nil.
`#intern, #to_sym`	Преобразовать строку в объект типа Symbol
`#length, #size`	Получить размер строки в символах.

Имя метода	Назначение

`#lstrip!, #lstrip, #rstrip!, #rstrip, #strip!, #strip`	Удалить пустые символы слева, справа или с обеих концов строки.
	`"  hello  ".strip  #=> "hello"`
	`"\tgoodbye\r\n".strip  #=> "goodbye"`

`#=~, #match`	Найти совпадения в строке по шаблоку регулярного выражения

`#next!, #next, #succ!, #succ`	Найти следующую по таблице символов комбинацию:
	`"abcd".succ  #=> "abce"`
	`"THX1138".succ  #=> "THX1139"`
	`"<<koala>>".succ  #=> "<<koalb>>"`
	`"1999zzz".succ  #=> "2000aaa"`
	`"ZZZ9999".succ  #=> "AAAA0000"`
	`"***".succ  #=> "**+"`

`#reverse!, #reverse`	Обратить последовательность символов в строке.
`#scan`	Выделить из строки массив строк, соответствующих шаблону. a = "cruel world" a.scan(/\w+/) #=> ["cruel", "world"] a.scan(/.../) #=> ["cru", "el ", "wor"] a.scan(/(...)/) #=> [["cru"], ["el "], ["wor"]] a.scan(/(..)(..)/) #=> [["cr", "ue"], ["l ", "wo"]]
`#slice!, #slice, #[]`	Выделить из строки подстроку, соответвующую шаблону или индексам.
`#split`	Разбить строку по указанному шаблону-разделителю. По-умолчанию символами разделителями являются пробелы, табуляции и пр.
`#squeeze!, #squeeze`	Сжать строку, удалив повторы указанных символов
`#sum`	Подсчитать контрольную сумму с указанным основанием.
`#to_f, #to_i, #to_c`	Преобразовать строку в число типа Float, Integer, Complex, соответственно.

Имя метода	Назначение

`#upto`	Выполнить последовательно изменение от начальной до конечной строки.
	`"a8".upto("b6") {|s| print s, ' ' }`
	`for s in "a8".."b6"`
	`  print s, ' '`
	`end`
	`# a8 a9 b0 b1 b2 b3 b4 b5 b6`
	`# a8 a9 b0 b1 b2 b3 b4 b5 b6`

Также приведем несколько простых примеров использования строк.

Создание строки:

```
a = "hello there"
```

Получение символа по индексу или заданное количество, начиная с указанного индекса:

```
a[1]                    #-> "e" (индексы начинаются с нуля)
a[1,3]                  #-> "ell"
```

Регулярное выражение над строкой:

```
a[/[aeiou](.)\1/, 0]    #-> "ell"
```

```
a = "hello ";
```

Конкатенация строк:

```
a << "world"    #-> "hello world"
```

Добавление символа по его коду:

```
a << 33         #-> "hello world!"
```

Создание новой строки как объединение указанных:

```
"Hello from " + self.to_s    #-> "Hello from main"
```

Сокращенная форма для строки форматного вывода:

```
"%-5s: %08x" % [ "ID", self.id ]    #-> "ID   : 200e14d6"
```

Вставка выражений в строку осуществляется с использованием конструкции #{Ruby-выражение}. Синтаксически видим здесь форму однострочного блока, а, вспоминая, что Ruby всегда возвращает значение последнего выражения, понимаем, что таким выражением может быть что угодно, включая простое обращение к некоторому объекту по имени константы или переменной. Обратите внимание на то, что для применения этой конструкции

строковый литерал должен быть создан либо с помощью двойных кавычек, либо с помощью заглавной буквы Q:

```
a = 1; b = 4
puts "The number #{a} is less than #{b} "
# The number 1 is less than 4
```

Приведем еще несколько примеров работы со строками из электронного учебника http://rubymonk.com/learning/books/1/chapters/5-strings/lessons/31-string-basics (обратите внимание, что методы #upcase, #downcase не работают для UNICODE-символов):

```
"Some string".include? 'string'
"Ruby is a beautiful language".start_with? "Ruby"
"I can't work with any other language but Ruby".end_with? 'Ruby'
"I am a Rubyist".index 'R'
'i am in lowercase'.upcase #=> 'I AM IN LOWERCASE'
'This is Mixed CASE'.downcase
"ThiS iS A vErY ComPlEx SenTeNcE".swapcase
'Fear is the path to the dark side'.split
'Ruby' + 'Monk'
"Ruby".concat("Monk")

"I should look into your problem when I get time".sub('I','We')

"I should look into your problem when I get time".gsub('I','We')
```

2.8. Регулярные выражения

Регулярные выражения широко применяются в скриптовых языках программирования, поскольку позволяют в компактной форме написать шаблон для поиска, замены и рассечения строки. При этом несущественно, какой размер имеет строка. Регулярное выражение является объектом класса Regexp. Приведем несколько примеров из Викиучебника «Регулярные выражения в строках» (см. https://ru.wikibooks.org/wiki/Ruby/Подробнее_о_строках):

```
"Жыло-было шыбко шыпящее жывотное".gsub(/(Ж|Ш|ж|ш)ы/){ $1 + "и" }

    #=> "Жило-было шибко шипящее животное"

"Жыло-было шыбко шыпящее жывотное".gsub(/([ЖШжш])ы/){ $1 + "и" }

    #=> "Жило-было шибко шипящее животное"
```

Скобки внутри регулярного выражения, ограниченного символами «/.../», означают группу. Внутри блока для метода gsub, который осуществляет замену найденного символа,

использована переменная $1, имеющая значение, равное первой группе соответственно. Подставляемое методом gsub значение определяется результатом операции **$1 + "и"** внутри блока, но здесь методом будет заменено все выражение целиком, т. е. группа и буква «ы».

Массив всех русских слов в тексте:

```
"Раз, два, три!".scan(/[А-Яа-я]+/)     #=> ["Раз", "два", "три"]
```

Все знаки препинания:

```
"Раз, два, три!".scan(/[, \.;:!]+/)     #=> [", ", ", ", "!"]
```

В приведенных примерах регулярных выражений символ «|» означает наличие любого из символов последовательности, символы внутри квадратных скобок [Жшжш] — единственный символ этой группы, вариант написания [А-Яа-я] — диапазон всех букв от А до Я и от а до я, однако допустима лишь одна буква.

Использование якоря (символа начала последовательности «^»):

```
str = “abc\ndef\nghi”
/def/ =~ str  # => 4
/^def/ =~ str # => 4
```

Приведем несколько примеров из документации по классу Regexp (https://ruby-doc.org/core-2.7.0/Regexp.html):

```
# 'haystack' не содержит шаблон 'needle',

#    следовательно, нет совпадения.
/needle/.match('haystack') #=> nil
# 'haystack' содержит шаблон 'hay', следовательно, есть совпадения
/hay/.match('haystack')    #=> #<MatchData "hay">

# Шаблон использует просмотр вперед и назад (lookahead,lookbehind)

# на наличие тэгов <b></b>, однако сами тэги
# не включаются в результат совпадения
/(?<=<b>)\w+(?=<\/b>)/.match("Fortune favours the <b>bold</b>")
    #=> #<MatchData "bold">

#Склейка и размножение

s = 'a' * 25 + 'd' 'a' * 4 + 'c'
    #=> "aaaaaaaaaaaaaaaaaaaaaaaaadadadadac"
/(b|a+)*\/ =~ s #=> 0
/(b|a+)*c/ =~ s #=> 32
```

Для регулярных выражений существуют квантификаторы (определяют кратность):

• * — нуль или более раз;
• + — один или более раз;
• ? — нуль или один раз;
• {n} — точно n раз;
• {n,} — n или более раз;
• {,m} — m или менее раз;
• {n,m} — не менее n и не более m раз.

Якори (определяют область применения выражения):

• ^ — начало строки;
• $ — конец строки;
• \b — граница слова;
• \B — не граница слова;
• (?=pat) — позитивный просмотр вперед. Найденная последовательность соответствует, но не включает pat;
• (?!pat) — негативный просмотр вперед. Найденная последовательность не соответствует и не включает pat;
• (?<=pat) — позитивный просмотр назад.

Для отладки регулярных выражений рекомендуется использовать интернет-ресурс https://rubular.com/

2.9. Операции с числами

В Ruby существуют следующие классы для работы с числами:

• Fixnum и Bignum — целые числа, меньше и больше 2^{30}, соответственно. Сейчас заменены типом Integer во всех случаях;
• Float — числа с плавающей запятой;
• BigDecimal — дробные числа;
• Rational — рациональные числа;
• Matrix — работа с матрицами;
• Complex — комплексные числа;
• Prime — класс для порождения простых чисел.

2.9.1. Числовые литералы

Примеры литералов:

```
237   # число без знака
+237  # число 237 без знака
-237  # отрицательное число
105327912 # число в десятичной записи
```

```
105_327_912 # то же число в бухгалтерском формате
0b101110 # двоичное число
0101110  # восьмеричное число
01234    # восьмеричное число
0xabcd # шестнадцатеричное число

# вещественные числа (для примера физические величины округлены)

3.14         # число Пи
6.02e23      # число Авогадро
6.626068e-34 # Постоянная Планка
```

Минимальное и максимальное значения чисел с плавающей точкой:

```
Float::MIN
Float::MAX
```

2.9.2. Основные операции над числами

В Ruby поддерживаются традиционные арифметические операции. Остановимся лишь на некоторых операциях.

Возведение в степень:

```
a = 64**2  # 4096
d = 64**-1 # 0.015625
```

Деление:

```
3 / 4      # 0! Целочисленное деление!
3 / 4.0    # 0.75
```

Явное приведение к плавающей точке:

```
x = x.to_f / y
```

Округление:

```
pi = 3.14159
puts pi.round  # 3

temp = -47.6
puts temp.round # -48
```

2.9.3. Форматирование вывода

Для форматирования вывода может быть применен метод «%» класса String.

Примеры:

```
"%05d" % 123                              #=> "00123"
"%-5s: %08x" % [ "ID", self.object_id ]   #=> "ID   : 200e14d6"
```

Формат записи в форматной строке соответствует функции printf из C/C++.

Примеры других способов преобразования чисел в строку:

```
x=123
x.to_s(2) # 1111011
print "With #{x} values\n";
printf ("%8.2f", x/456.26)
```

2.9.4. Очень большие числа

Очень большие числа представлены классом Bignum. Числа этого класса будут автоматически получены, если результат выполнения операции превышает разрядность Fixnum. Класс Bignum не имеет формального ограничения на размер чисел. Размер числа ограничивается лишь доступной оперативной памятью.

Пример. Выведем число и его тип, последовательно умножая его:

```
x = 1000000
4.times do
  puts x.to_s() + "\t" + x.class.to_s()
  x *= x
end
```

Для Ruby версий ранее 2.4 получим следующий результат:

```
1000000 Fixnum
1000000000000   Bignum
100000000000000000000000       Bignum
10000000000000000000000000000000000000000000000000       Bignum
```

Для Ruby версии 2.4 и далее, результат выглядит как:

```
1000000 Integer
1000000000000     Integer
100000000000000000000000    Integer
10000000000000000000000000000000000000000000000000    Integer
```

То есть современные версии Ruby имеют только один тип для целых чисел, но их размер не ограничен.

2.10. Символы и диапазоны

Символы в Ruby — это экземпляры класса Symbol. По своему смыслу они близки константным строкам (к ним не применимы операции над строками), но принципиально отличаются от них тем, что на одну последовательность символов в оперативной памяти будет создан только один экземпляр. Литерал символа начинается со знака «:».

Пример создания символов:

```
array = ["string", "string", "string", :string, :string, :string]
```

В результате выполнения примера будет создано 3 объекта String, содержащих "string" и единственный объект класса Symbol, содержащий string. Несмотря на то, что :string выглядит, как идентификатор, идентификатором он не является, а знак «:» является лишь признаком того, что это символ.

Допустимы также символы следующего вида:

```
sym = :"This is a symbol"
```

Строки могут быть преобразованы в символы, а символы могут быть преобразованы в строки:

```
a = "somestr"
b = :somestr
a == b.to_str #true
b == a.to_sym #true
```

Диапазоны чаще всего применяются для формирования числовых последовательностей. Диапазоны представляют собой объекты класса Range. Примеры числовых диапазонов:

```
r1 = 1..3  #закрытый диапазон
r2 = 1...3 #открытый диапазон (не включает последнюю точку)
```

Обход по диапазону:

```
r1.each {|x| puts x}
(4..7).each {|x| puts x}
puts r1.first, r1.last
```

Диапазоны со строками:

```
a = "a".."z"
puts a.include? "b"              # true
```

Следует отметить, что в Ruby версий 1.8 и 1.9 по-разному реализовано формирование строковых диапазонов:

```
puts a.include? "bb"            # false для 1.9 и true для 1.8
puts ("a".."zz").include?("bb") # true
puts ("2".."5").include?("28")  # false для 1.9 и true для 1.8
```

Существуют и другие методы для работы с диапазонами. См. https://ruby-doc.org/core-2.7.0/Range.html

2.11. Консольный ввод-вывод

Для взаимодействия с консолью могут быть использованы глобальные объекты STDIN, STDOUT класса IO, связанные со стандартными потоками ввода/вывода. При этом допустимо опускать STDIN и STDOUT и применять следующие методы непосредственно:

• puts foo — вывод foo как строки, что эквивалентно puts foo.to_s;

• print — вывод строки без \n в конце;

• printf — аналогичен C printf;

• p foo — вывод значения, что эквивалентно puts foo.inspect.

Для объекта STDIN доступны методы:

• gets — помещение результата ввода строки данных в переменную $_ и возвращение строки;

• getc — чтение одного символа.

Пример консольного ввода-вывода:

```
# coding: utf-8

a = [ [1,2,3], [3,4,5], [7,8,9] ]
b = [ 1,2,3 ]
c = 1

print "-----\n"
a.each_index {|e| print a[e], "\t"  }
# выведено [1, 2, 3]        [3, 4, 5]        [7, 8, 9]
print "\n-----\n"

for i in 0..2
  for j in 0..2
    a[i][j]=a[j][i]
  end
end

print "\n-----\n"
print a, "\n" # выведено [[1, 3, 7], [3, 4, 8], [7, 8, 9]]
print a[0][0], "\n"
print b[1], "\n"
print c, "\n"
```

```
print "-----\n"
s = "введите строку"

str = gets
print str

file = File.open('out.txt', 'w')
file.write( s )
# преобразуем строки в различные кодировки
file.write( str.encode("CP866") )
file.write( s.encode('windows-1251') )
file.close
```

Обратите внимание на то, что кодировка консоли может не соответствовать кодировке файла, поэтому необходимо правильно преобразовывать строки при вводе и выводе. Для реализации преобразования служит метод String#encode. По-умолчанию выполняется трансляция строк в UNICODE в соответствии с настройками текущих локалей операционной системы.

2.12. Файловые операции

Для работы с файлами существует класс File. Приведем пример из документации по классу File (см. https://ruby-doc.org/core-2.7.0/File.html):

```
f = File.new("out", "w")
f.write("1234567890")     #=> 10
f.close                   #=> nil
File.truncate("out", 5)   #=> 0
File.size("out")          #=> 5
```

Так же, как и в языках C/C++, файлы следует закрывать после окончания их использования с помощью метода close. В первую очередь это относится к операции записи. Однако в Ruby существует специальная форма открытия файла с блоком, позволяющая не вызывать close.

Ниже приведены несколько примеров программ на русском языке (http://rubydev.ru/2011/06/ruby_file_io_api_pt1/), на английском языке (http://marcosccm.com/posts/ruby-file-io-primer-part-1-the-file-class):

```
f = File.new("lib/file.rb")
while line = f.gets
  puts line
end
f.close
```

```
f = File.new("lib/file.rb")
f.each do |line|
  #делаем что-то со строкой
end
f.close
```

Открытие файлов с автоматическим закрытием:

```
File.open("1.txt", "w") do |f|
  f.puts "что-то записывается в файл"
end

File.open(__FILE__, "r") do |f|
  while line = f.gets
    puts line
  end
end
```

Обратите внимание на переменную __FILE__, которая хранит имя текущего файла. Этот фрагмент кода распечатает файл, в котором фрагмент содержится.

Обработка бинарного файла:

```
open('binary.dat', 'rb') { |f| f.each_byte { |b| puts b } }
```

Позиционирование в файле осуществляется с помощью метода seek. Первый байт имеет номер 0. Пример:

```
f = File.new("lib/file.rb")
f.seek(20) # перейти к 20-му символу
line = f.gets
puts line
f.close
```

С другими примерами использования Ruby для работы с бинарными файлами можно ознакомиться на сайте https://www.rubyguides.com/2017/01/read-binary-data/.

Приведем еще несколько примеров полезных операций с файлами.

Прочитать весь файл и получить строку:

```
str = File.read 'filename.txt'
```

Прочитать все строки и сохранить в виде массива:

```
array = File.readlines 'filename.txt'
```

Проверить наличие файла:

```
File.exist? 'filename.txt' # => true or false
```

Проверить, является ли директорией:

```
File.directory?(file_name) # => true or false
```

2.13. Массивы

Массивы в Ruby являются объектами класса Array и могут быть созданы несколькими способами (см. https://ru.wikibooks.org/wiki/Ruby/Справочник/Array; https://ru.wikibooks.org/wiki/Ruby/Подробнее_о_массивах). Доступ к элементам массива может быть осуществлен по порядковому номеру начиная с нуля. Поскольку любой класс в Ruby является потомком класса Object, массивы могут хранить любые объекты в любой комбинации.

Примеры использования массивов приведены ниже. Примеры заимствованы из Викиучебников (https://ru.wikibooks.org/wiki/Ruby/Спра-вочник/Array; https://ru.wikibooks.org/wiki/Ruby/Подробнее_о_массивах).

Создание массивов с помощью литерала:

```
array = ["a", "b", "c", "d", "e"]
```

Обращение к элементу по индексу:

```
array[array.size - 2]  #=> "d"
array[-2]              #=> "d"
```

Многомерные массивы:

```
[[1], [2, 3], [4]]   # разная длина элементов-массивов
[[1, 2], [3, 4]]     # одинаковая длина

# двумерный массив

[["прива", "Привет"], ["пока", "Всего хорошего"]]

# гибрид двух-трех-мерного массива

[["прива", "Привет"], [1, ["пока", "Всего хорошего"]]]
```

Создание массивов методом класса new:

```
Array.new(size=0, obj=nil)

Array.new(array)
Array.new(size){|index| block }
```

Проверка не пустого массива (по стилю программирования на языке Ruby рекомендуется применять метод, который не потребует в условии использовать логическое отрицание):

```
array = [1, 2, 4]
```

```
array.size > 0      #=> true
array.length > 0    #=> true
array.empty?        #=> false
array.any?          #=> true
```

Поиск совпадения:

```
array = [1, 2, 3, 4, 5, 6, 7]
array.include?(5)  # true
```

Определение максимального/минимального элемента:

```
["у", "попа", "была", "собака"].max
#     => "у" максимальный по значению

["у", "попа", "была", "собака"].max_by{ |elem| elem.size }
#     => "собака" максимальный по размеру строки
```

Методы .min и .min_by работают аналогично:

```
["у", "попа", "была", "собака"].min
#     => "была" минимальный по значению
["у", "попа", "была", "собака"].min_by{ |elem| elem.size }

#     => "у"    минимальный по размеру строки
```

Упорядочение массивов возможно с помощью методов sort или sort_by:

```
["у", "попа", "была", "собака"].sort
#      => ["была", "попа", "собака", "у"] сортировка по значению
["у", "попа", "была", "собака"].sort_by{ |elem| elem.size }
#  => ["у", "попа", "была", "собака"] сортировка по размеру строки
```

Упорядочение для двумерных массивов:

```
[[1,0], [16,6], [2,1], [4,5],[4,0],[5,6]].sort_by {|elem| elem[1]}
#   => [[1, 0], [4, 0], [2, 1], [4, 5], [16, 6], [5, 6]]
#       сортировка "внешних" элементов по значению "внутренних"
[[1,0], [16,6], [2,1], [4,5],[4,0],[5,6]].sort_by {|elem| elem[0]}
#   => [[1, 0], [2, 1], [4, 0], [4, 5], [5, 6], [16, 6]]
```

Сложение/вычитание массивов:

```
[1, 2, 3, 4] + [5, 6, 7] + [8, 9]  #=> [1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 1, 2, 2, 3, 3, 3, 4, 5] - [1, 2, 4]    #=> [3, 3, 3, 5]
```

Удаление дубликатов:

```
[1, 2, 3, 4, 5, 5, 6, 0, 1, 2, 3, 4, 5, 7].uniq
#      => [1, 2, 3, 4, 5, 6, 0, 7],
```

Размножение:

```
["1", "2", "3"] * 2   #=> ["1", "2", "3", "1", "2", "3"]
[1, 2, 3, 4] * 2            #=> [1, 2, 3, 4, 1, 2, 3, 4]
[1, 2, 3, 4] + [1, 2, 3, 4] #=> [1, 2, 3, 4, 1, 2, 3, 4]
```

Следует отметить некоторые особенности применения индексов элементов. Как уже упоминалось, нумерация элементов начинается с нуля, например:

```
a = [ "a", "b", "c", "d", "e" ]
a[0]        #=> ["a"]
```

При выходе за границы массива или при обращении к элементу, который не был инициализирован, получим nil:

```
a[6]            #=> nil
```

Если необходимо получить диапазон элементов массива, можно использовать форму, указывающую первый элемент и количество возвращаемых элементов:

```
a[1, 2] #=> [ "b", "c" ]
```

или формулу, указывающую номера элементов, заданные диапазоном:

```
a[1..3] #=> [ "b", "c", "d" ]
```

Ruby имеет отрицательные номера элементов. Значение –1 соответствует последнему элементу массива. Так же как и при использовании положительных индексов, диапазон требуемых элементов можно указать номером элемента, отсчитывая с конца массива, и количеством элементов:

```
a[-4, 2]        #=> [ "b", "c" ] с 4-го с конца, 2 элемента
```

а также диапазоном, начиная с указанного элемента с начала массива и до указанного с конца массива:

```
a[3..-1]        #=> [ "d", "e" ] с 4-го и до конца
```

Аналогично массивам возможно обращение к элементам строки:

```
str = "1234567890"
str[4..-1]      #=> "567890"
```

Более подробно изучить работу с массивами можно, используя документацию по классу Array (см. https://ruby-doc.org/core-2.7.0/Array.html).

Для работы с элементами массивов существуют специальные методы: each, each_with_index, find, index и пр., которые позволяют не использовать традиционные циклы.

2.14. Ассоциативные массивы

Ассоциативные массивы являются структурами, позволяющими хранить данные аналогично массивам. Однако доступ к элементам осуществляется с помощью ключа любого типа. Ассоциативные массивы (они же хеши) являются объектами класса Hash. Как и простые массивы, ассоциативные массивы могут хранить объекты любых классов в любых комбинациях, включая вложенные ассоциативные массивы.

Создание ассоциативного массива с помощью литерала:

```
hash = {5=>3, 1=>6, 3=>2}
hash[5]                 #=> 3
hash[2]                 #=> nil  это значит, что объект отсутствует
hash[3]                 #=> 2
```

Создание ассоциативного массива с помощью конструктора (см. https://ru.wikibooks.org/wiki/Ruby/Справочник/Hash):

```
h = Hash.new("Go Fish") # создается объект по-умолчанию! Иначе nil
h["a"] = 100
h["b"] = 200
puts h["a"]           #-> 100
puts h["c"]           #-> "Go Fish"

# Изменяется единственный объект по умолчанию
puts h["c"].upcase!   #-> "GO FISH"
puts h["d"]           #-> "GO FISH"
puts h.keys           #-> ["a", "b"]
```

Обратите внимание на то, что в приведенном примере при обращении по ключу, который не был добавлен, будет возвращен объект по умолчанию. И здесь он был единственным — строка "Go Fish"!

Создание нового объекта при каждом обращении по несуществующему ключу:

```
# Создается новый объект по умолчанию каждый раз
h = Hash.new { |hash, key| hash[key] = "Go Fish: #{key}" }
puts h["c"]           #-> "Go Fish: c"
puts h["c"].upcase!   #-> "GO FISH: C"
puts h["d"]           #-> "Go Fish: d"
puts h.keys           #-> ["c", "d"]
```

Рассмотрим еще несколько примеров создания и использования ассоциативных массивов (см. https://ru.wikibooks.org/wiki/Ruby/Подробнее_об_ассоциативных_массивах).

Создадим ассоциативный массив из одномерного массива. В наличии индексный массив, где ключ и значение записаны последовательно. Тогда мы применим связку методов * и Hash[]:

```
array = [1, 4, 5, 3, 2, 2]
Hash[*array]  #=> {1=>4, 5=>3, 2=>2}
```

Элементы, стоящие на четной позиции (в данном случае: 1, 5 и 2), стали ключами, а элементы, стоящие на нечетной позиции (т. е. 4, 3 и 2), — значениями.

Создадим ассоциативный массив из двумерного массива. Если имеется массив в формате [["ключ_1", "значение_1"], ["ключ_2", "значение_2"], ["ключ_3", "значение_3"], …], то его надо привести к одномерному (.flatten), после чего задача будет сведена к предыдущей:

```
array = [[1, 4], [5, 3], [2, 2]]
Hash[*array.flatten]          #=> {1=>4, 5=>3, 2=>2}
```

Для получения отдельно массива ключей или массива значений существуют методы keys и values:

```
{1=>4, 5=>3, 2=>2}.keys       #=> [1, 2, 5]
{1=>4, 5=>3, 2=>2}.values     #=> [4, 3, 2]
```

Имеются специальные формы литералов для хешей, у которых ключами являются символы. Например, следующие записи эквивалентны:

```
{ :a => 1, :b => 2};  {a: 1, b: 2};
```

2.15. Множества

Множества могут рассматриваться как частный случай массивов (см. https://ru.wikibooks.org/wiki/Ruby/Подробнее_о_массивах).

Операция объединения имеет вид:

```
[1, 2, 3, 4, 5, 5, 6] | [0, 1, 2, 3, 4, 5, 7]
#     => [1, 2, 3, 4, 5, 6, 0, 7]
```

Объединение получается следующим образом. Сначала массивы сцепляются:

```
[1, 2, 3, 4, 5, 5, 6, 0, 1, 2, 3, 4, 5, 7]
```

Затем, начиная с первого элемента, удаляются элементы, которые уже встречались. После зачистки получается настоящее логическое объединение.

Операция пересечения:

```
[1, 2, 3, 4, 5, 5, 6] & [0, 2, 1, 3, 5, 4, 7]  #=> [1, 2, 3, 4, 5]
```

При пересечении двух массивов из первого удаляются все элементы, отсутствующие во втором. При этом относительный порядок остающихся элементов первого массива сохраняется.

Ruby также имеет специальный класс Set. В отличие от обычных массивов класс Set использует класс Hash, интерпретируя переданный набор данных как ключи. Поэтому доступ по ключу происходит многократно быстрее.

Пример из документации Ruby:

```
require 'set'
s1 = Set.new [1, 2]                   # -> #<Set: {1, 2}>
s2 = [1, 2].to_set                    # -> #<Set: {1, 2}>
s1 == s2                              # -> true
p s1.add("foo")                       # -> #<Set: {1, 2, "foo"}>
p s1.merge([2, 6])                    # -> #<Set: {6, 1, 2, "foo"}>
p s1.subset? s2                       # -> false
p s2.subset? s1                       # -> true
```

2.16. Методы и блоки

Поскольку Ruby — это полностью объектный язык, функций без класса или объекта не существует. С точки зрения терминологии в Ruby нет функций — есть методы. Методы, у которых при декларации опущен класс, станут методами класса Object.

Пример декларации методов с помощью ключевого слова def:

```
def func1(x)
  y=x
  y=x+5 if x<10
  y=x*5 if x>15
  return y
end
def func2(x)  x*x; end
def func3 x;  x+2; end
```

Любой метод возвращает результат. В func2 в качестве результата возвращается результат последней операции x*x (здесь

она единственная), в func1 — значение 'y' с явным вызовом return. Обратите внимание на то, что если не указан return y или просто y, то возвращаемым результатом в данном коде был бы результат сравнения x>15, но не y!

Блоки являются важной частью Ruby. Помимо выполнения функции составного оператора они позволяют применить конструкцию yield, что, по сути, дает возможность внедрить свой код в ранее созданные методы, причем с получением значения, которое следует обработать.

Блоки могут не использовать переменные, например,

```
10.times {print '*'} # => **********
```

или использовать переменную (их количество определяется методом, к которому присоединен блок), причем передаваемые переменные декларируются внутри символов «|...|»:

```
10.times {|i| print i} # => 0123456789
```

Поясним сущность блока с помощью следующего выражения:

```
File.open('filename'){ |f| f.puts 'что-то записывается в файл' }
```

Оно эквивалентно (на уровне псевдокода) выражению

```
f = File.open(file, 'r')
  f.puts 'что-то записывается в файл'
f.close
```

Рассмотрим другой пример. Определим метод test_func, единственное действие которого заключается в запуске цикла с числом повторений, указанным в качестве аргумента. Тело этого цикла будет активировать блок.

Пример:

```
def test_func (x)
  for i in 0...x do
    yield i+1
  end
end
test_func(5) { |n| s="#{n}:"; n.times {s += "*"}; puts s; }
```

Результат выражения:

```
1:*
2:**
3:***
4:****
5:*****
```

Разберем пример подробнее. Вызов метода test_func(5) содержит аргумент 5 и блок {|n| s="#{n}:"; n.times {s += "*"}; puts s;}. В тексте блока содержится декларация |n|, которая обеспечивает передачу значения переменной их метода test_func. Остальная часть блока представляет собой код, который будет вызван из метода test_func. В самом методе test_func имеется специальная конструкция yield, которая обеспечивает передачу параметра внутрь блока и передачу управления к коду блока, расположенного в вызывающей метод test_func части программы. Процесс выполнения кода иллюстрирует схема, приведенная на рис. 2.2.

В результате подстановки блока в тело основного метода получается следующий эквивалентный код:

```
def test_func (x)
  for i in 0...x do
    n=i+1; s="#{n}:";  n.times {s += "*"}; puts s;
  end
end
test_func(5)
```

По сути, блок представляет собой анонимный метод, вызываемый как функция обратного вызова, если говорить в терминах языка C.

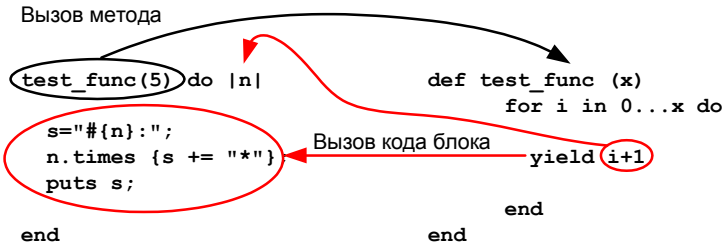

Рис. 2.2. Вызов кода блока из вызванного метода

Обратите внимание на то, что переменные, которые объявлены внутри |...|, являются ссылками на какие-то объекты, переданные из вызванного метода, а любые попытки изменить значения ссылок никоим образом не изменят состояние исходных объектов внутри метода. Выполним следующий код:

```
def test_str(str)
  yield str;
  puts str;
end
test_str('123') { |s| s='456'; puts s }
```

Получим сообщение 456 123, но не 456 456, поскольку переменная s была переставлена на другой объект, но метод, который вызвал yield, ничего о ней не знает. При этом для переменных блока возможно применение методов, модифицирующих объект, таких как **sub!**, поскольку str и s указывают на один и тот же объект.

В тех случаях, когда необходимо изменить значение, метод, вызывающий блок, должен анализировать возвращаемое yield значение:

```
def test_str(str)
   s=yield str;
   puts s;
end
test_str('123') { |s| puts s; '456' }
```

Получим '123 456', поскольку yield вернет значение последнего выражения внутри блока.

Обратите внимание на то, что переменные внутри |...| всегда локальны для блока и заменяют одноименные переменные, которые были созданы на уровень выше.

Говоря о блоках, следует упомянуть о специальном классе Proc. Этот класс позволяет создавать код как объект, который будет в дальнейшем выполнен. Рассмотрим простой пример метода, имеющего блок, однако для того, чтобы явно управлять выполнением переданного кода, в декларации метода добавим аргумент &block:

```
def g2(x, &block)
  # выводим тип объекта блока
  p block
  # запускаем код на выполнение через метод call
  block.call(x) * block.call(x)
end

#традиционный вариант блока
puts g2(7){|x| x+3} # => #<Proc:0x28fb5b8@ruby_func.rb:9>

#создаем объект с кодом, используя синтаксис блока
pr = Proc.new{|x| x+3 }
puts g2(7, &pr) # => #<Proc:0x28fb300@ruby_func.rb:12>

# создаем объект lambda – функция как код.
```

```
lm = ->(x) { x+3 }
puts g2(7, &lm) # => #<Proc:0x28fb048@ruby_func.rb:14 (lambda)>
```

Как видим, все три варианта передачи кода позволили выполнить код и подставить его в качестве блока. Причем простейший вариант блока — это тот же экземпляр класса Proc, но отличается от второго варианта тем, что не существует именованной ссылки на него. Lambda — это функция, созданная как объект.

Контрольные вопросы и задания

1. Укажите, в чем различие методов p, puts, print.

2. Укажите, что будет выведено на экран:

```
p '*'*2*1+'*'
```
?

Для проверки используйте irb.

3. Укажите, что будет выведено на экран:

```
[nil, 0, 1, true, false, '', '123'].each do |i|
    puts i if i
end
```

Для проверки используйте irb.

4. Укажите, что будет выведено на экран:

```
p [nil, 0, 1, true, false, '', '123'].select {|i| not i }
```

Для проверки используйте irb.

5. Могут ли в массиве Ruby одновременно храниться объекты числового и строковых типов?

6. Что означает следующая запись:

```
a[3..-1]
```
?

7. Что такое Symbol и в чем его отличие от String?

8. Каково назначение регулярных выражений?

9. Что означают следующие записи:

```
{a => 1, b => 2};
```

```
{:a => 1, :b => 2};
{a: 1, b: 2}; ?
```

10. Что такое блок? Приведите пример использования и создания метода с блоком.

3. ОБЪЕКТНЫЕ СРЕДСТВА ЯЗЫКА

3.1. Классы

Ruby является чистым объектным языком, поскольку не существует простых типов, экземпляры которых не являются объектами. В отличие от C++ его объектная модель намного полнее. Одни и те же классы могут быть распределены по разным модулям. Любой объект может получить полную информацию о классе, к которому он принадлежит. Более того, любой объект может переопределить свои методы и свойства при выполнении программы. Тело класса может быть распределено по нескольким модулям. Базовым классом в Ruby является Object.

Переменные класса (одинаковые и доступные всем экземплярам данного класса и его подклассов) всегда обозначаются символами **@@**, например **@@**name.

Переменные экземпляра класса (индивидуальные для каждого экземпляра) всегда обозначаются символом **@**, например **@**a.

Константы класса всегда пишутся ПРОПИСНЫМИ буквами.

Методы класса вызываются только с именем класса, методы экземпляра класса — через переменную или иным способом, полученным экземпляром класса.

Имя класса является константой, поэтому может начинаться только с заглавной буквы.

3.1.1. Конструкторы

В Ruby нет конструктора как такового. Любой объявленный класс является экземпляром класса Class, а метод new, который создает объект, принадлежит классу Class. Поэтому если имеется класс Name (при этом он является константой, указывающей на объект типа Class, что легко проверить через вызов **puts Name.class**),

то вызов Name.new создает объект класса Name, т. е. выделяет для него память. Далее метод new вызывает метод инициализации initialize и передает ему параметры, с которыми он сам был вызван. Класс может иметь только один метод initialize. Если необходимы несколько конструкторов, то следует определить дополнительные статические методы класса, возвращающие новый объект.

Рассмотрим пример из книги [1]. Класс ColoredRectangle имеет метод initialize, инициализирующий все его внутренние переменные. Дополнительно созданы методы класса white_rect, red_square, которые принимают необходимое количество аргументов:

```
class ColoredRectangle

     def initialize(r, g, b, s1, s2)
      @r, @g, @b, @s1, @s2 = r, g, b, s1, s2
     end
     # определяем статический метод через имя класса
     def ColoredRectangle.white_rect(s1, s2)
      new(0xff, 0xff, 0xff, s1, s2)
     end
     # определяем статический метод через ссылку на класс
     def self.red_square(s)
      new(0xff, 0, 0, s, s)
     end
     # переопределяем inspect, используемый в методе p
     def inspect
      "#@r #@g #@b #@s1 #@s2"
     end
end

a = ColoredRectangle.new(0x88, 0xaa, 0xff, 20, 30)
b = ColoredRectangle.white_rect(15,25)
c = ColoredRectangle.red_square(40)

p a,b,c
```

С точки зрения C++ методы white_rect и red_square не являются конструкторами (там они могли быть созданы как static-методы), но возвращают они новый объект с заданными свойствами, т. е. по поведению соответствуют конструктору.

Обратите внимание на то, что white_rect и red_square по-разному объявлены: первый — конкретно для класса ColoredRectangle, второй — для **self**, т. е. для объекта в контексте текущего выполняемого кода. Здесь эти записи равнозначны и декларация класса эквивалентна последовательному выполнению кода внутри **class Name … end**. Поэтому self внутри этого блока указывает на объект класса Class, на который будет указывать

константа **Name** (или ColoredRectangle в данном случае). Вариант с self более универсален, в том числе для случая переноса кода или переименования класса.

Обратите внимание на следующую особенность. Декларация класса может содержать любой код. Например:

```
class ColoredRectangle
  3.times { |i| print i.to_s + '.' }
end
```

Выдаст на печать «**0.1.2.**» И естественный вопрос, почему это происходит. Ответ заключается в том, что в модели языка Ruby, слово «class» следует воспринимать как метод, который создает класс, а не как какое-то специальное служебное слово по аналогии с C++ или Java. Аргументом этого метода здесь является ColoredRectangle, то есть константа, которая получит ссылку на созданный объект класса. А тело этой декларации, по сути, представляет собой код со вполне обычным кодом на Ruby, который последовательно выполняется. Аналогично следует воспринимать и метод «def», назначением которого является создание методов с указанным именем. И в этом смысле, нет разницы между телом декларации def, class или module, поскольку во всех этих случаях код будет последовательно выполнен.

3.1.2. Атрибуты

В Ruby различают атрибуты уровня экземпляра и уровня класса. Атрибуты экземпляра класса (имеют префикс @) могут быть созданы явно при конструировании объекта или не явно при выполнении методов этого класса в момент первого присвоения или специальным методом. Поскольку атрибуты определяют внутреннее состояние объекта, целесообразно обращаться к ним только через специальные интерфейсные методы класса. Эти методы можно запрограммировать или с использованием специальных средств attr, или «вручную», как показано ниже:

```
class Person

     def name
      @name      # возвращает значение @name
     end
```

```
    def name=(x)
     @name = x # инициализирует @name
    end

    def age
     @age      # возвращает @age
    end

    # ...
end
```

Обратите внимание на то, что в явном виде атрибуты нигде не декларируются. Любое обращение к переменной с именем, у которого имеется префикс @ (@name) приводит к чтению или созданию соответствующего атрибута. При этом в полном согласии с концепцией недоступности внутреннего состояния (атрибутов) объекта для доступа извне предусмотрены специальные методы для получения и присвоения значений: **def name**, **def name= (val)**.

Существуют и более короткие способы определения этих методов. Специальный метод attr принимает в качестве параметра символ и создает соответствующий атрибут. Кроме того, он создает одноименный с ним метод чтения, а если необязательный второй параметр равен true, то и метод модификации.

```
class Person
 attr :name, true # Создаются @name, name, name=
 attr :age        # Создаются @age, age
end
```

Специальные методы attr_reader, attr_writer и attr_accessor принимают в качестве параметров произвольное число символов. Первый создает только методы чтения (для получения значения атрибута); второй — только методы установки, а третий — и то, и другое. Пример:

```
class SomeClass
 attr_reader :a1, :a2   # Создаются @a1, a1, @a2, a2
 attr_writer :b1, :b2   # Создаются @b1, b1=, @b2, b2 =
 attr_accessor :c1, :c2 # Создаются @c1, c1, c1=, @c2, c2, c2=
 # ...
end
```

Атрибуты для уровня класса декларируются с префиксом **@@**. Поскольку любой класс — это объект типа Class, то концептуально ясно, что этот объект и является местом хранения атрибутов уровня всего класса. Приведем пример из книги [1]:

```
class Metal
  @@current_temp = 70 # переменная, общая для всех экземпляров!

  attr_accessor :atomic_number

  def Metal.current_temp=(x)
   @@current_temp = x
  end

  def Metal.current_temp
   @@current_temp
  end

  def liquid?
   @@current_temp >= @melting
  end

  def initialize(atnum, melt)
   @atomic_number = atnum # атомный номер элемента
   @melting = melt        # температура плавления
  end
end

# создаем 3 объекта класса Metal
aluminum = Metal.new(13, 1236)
copper = Metal.new(29, 1982)
gold = Metal.new(79, 1948)

# устанавливаем общую температуру
Metal.current_temp = 1600

# смотрим, кто расплавился
puts aluminum.liquid? # true
puts copper.liquid?   # false
puts gold.liquid?     # false

# повышаем общую температуру
Metal.current_temp = 2100

# смотрим, кто теперь расплавился
puts aluminum.liquid? # true
puts copper.liquid?   # true
puts gold.liquid?     # true
```

Рассмотрим еще один интересный пример, в котором использованы средства Ruby для поиска экземпляров своего класса (см. http://juixe.com/techknow/index.php/2007/01/22/ruby-class-tutorial/):

```
class Person
  attr_accessor :fname, :lname

  def initialize(fname, lname)
    @fname, @lname = fname, lname
  end
```

```
  def to_s
    @lname + ", " + @fname
  end

  def self.find_by_fname(fname)
    found = nil
    #получаем список всех объектов указанного класса
    ObjectSpace.each_object(Person) { |o|
      found = o if o.fname == fname
    }
    # возвращаем последний найденный элемент
    found
  end
end

Person.new("Yukihiro", "Matsumoto")
Person.new("David", "Thomas")
Person.new("David", "Black")
Person.new("Bruce", "Tate")

 # Find matz!
puts Person.find_by_fname("Yukihiro")
```

В этом примере применен не самый оптимальный способ поиска, поскольку ObjectSpace.each_object(Person) перебирает все объекты класса Person. Пример приведен только для иллюстрации возможностей Ruby.

3.1.3. Наследование

В Ruby реализовано одиночное наследование. Для использования наследования предназначен знак «<».

Пример. Создадим класс для описания человека как такового:

```
class Person

  attr_accessor :name, :age, :gender

  def initialize(name, age, gender)

   @name, @age, @gender = name, age, gender
  end

  # ...

 end
```

Добавим класс Student как производный от человека:

```
 class Student < Person

  attr_accessor :idnum, :hours

  def initialize(name, age, gender, idnum, hours)
   # вызываем 'конструктор' предка
```

```
    super(name, age, gender)

    @idnum = idnum

    @hours = hours
  end
  # ...
end

# Создать два объекта.
a = Person.new("Dave Bowman", 37, "m")
b = Student.new("Franklin Poole", 36, "m", "000-13-5031", 24)
```

В методе #initialize класса потомка присутствует вызов метода инициализации предка super.

Из особенностей Ruby можно отметить возможность доступа к метаданным класса, в том числе возможность получить информацию о том, экземпляром какого класса является данный объект, включая всех его предков и примеси.

Пример доступа к метаданным:

```
s = "Hello"
n = 237
sc = s.class # String
nc = n.class # Fixnum
```

Примеры проверки типа:

```
n = 9876543210
flag1 = n.instance_of? Numeric # false
flag2 = n.kind_of? Bignum      # true
flag3 = n.is_a? Bignum         # true
flag3 = n.is_a? Integer        # true
flag4 = n.is_a? Numeric        # true
flag5 = n.is_a? Object         # true
flag6 = n.is_a? String         # false
flag7 = n.is_a? Array          # false
```

Обратите внимание на разницу в методах #is_a?, #kind_of? и #instance_of?. Первые два метода проверяют, что объект является экземпляром указанного класса или любого его потомка. То есть является видом. #instance_of? проверяет строгое соответствие типа.

Классы можно сравнить между собой. Большим будет тот класс, который является предком:

```
flag1 = Integer < Numeric # true
flag2 = Integer < Object  # true
flag3 = Object == Array   # false
flag4 = IO >= File        # true
flag5 = Float < Integer   # nil
```

3.1.4. Управление доступом к методам и экземплярам

Существует возможность определить методы как private или protected:

```
class Bank
  # определяем методы
  def open_safe
   # ...
  end

  def close_safe
   # ...
  end

  # а теперь делаем их недоступными извне
  private :open_safe, :close_safe

  def make_withdrawal(amount)
   if access_allowed
    open_safe
    get_cash(amount)
    close_safe
   end
  end

 # Остальные методы закрытые.
 private
  def get_cash
   # ...
  end

  def access_allowed
   # ...
  end
 end
```

Приведем пример класса, позволяющего сравнивать объекты по возрасту, но не получить точное значение возраста:

```
class Person
  def initialize(name, age)
   @name, @age = name, age
  end

  def <=>(other)
   age <=> other.age
  end

  attr_reader :name, :age # определим атрибуты
  protected :age          # а теперь закроем возраст
 end

 p1 = Person.new("fred", 31)
 p2 = Person.new("agnes", 43)
 compare = (p1 <=> p2) # -1
 x = p1.age            # Ошибка!
```

Следует обратить внимание на то, что запись **private** :open_safe, :close_safe однозначно говорит о том, что слово **private** является методом, а методы, которые необходимо скрыть, передаются по именам как символы. То же могло бы быть записано как **private** 'open_safe'.**to_sym**,..., однако эта запись не эффективна, поскольку требует явно создать объект типа String и лишь после этого явно его преобразовать в символ.

3.1.5. Константы и замораживание экземпляров

Весьма важной особенностью Ruby является раздельное существование понятий идентификаторов-констант и неизменяемых экземпляров классов. Эта особенность является следствие его динамической природы. Константа, как и переменная, всегда ссылается на какой-то экземпляр. Или на объект nil, если не были заданы другие. Однако экземпляр (объект) вполне автономен и не зависит от того, кто на него ссылается. Любой экземпляр может быть заблокирован на изменение с помощью метода #freeze. Пример:

```
str = "Тест "
str.freeze

begin
 str << "не пройден!" # Попытка модифицировать.
rescue => err
 puts "#{err.class} #{err}"
end

arr = [1, 2, 3]
arr.freeze

begin
 arr << 4 # Попытка модифицировать.
rescue => err
 puts "#{err.class} #{err}"
end

# Выводится:
# TypeError: can't modify frozen string
# TypeError: can't modify frozen array
```

Однако существует проблема, которая может быть проиллюстрирована следующими примерами:

```
str = "counter-"
str.freeze
str += "intuitive" # => "counter-intuitive"
```

```
arr = [8, 6, 7]
arr.freeze
arr += [5, 3, 0, 9] # => [8, 6, 7, 5, 3, 0, 9]
```

Причина того, что внешне объект изменяется, заключается в том, что результатом операции += или операции a = a+… является создание нового объекта, т. е. исходный объект остается заблокированным и неизменным, но переменная, которая указывала на него, указывает на новый объект!

Подведем итог по заморозке объектов. См. Рис. 3.1.

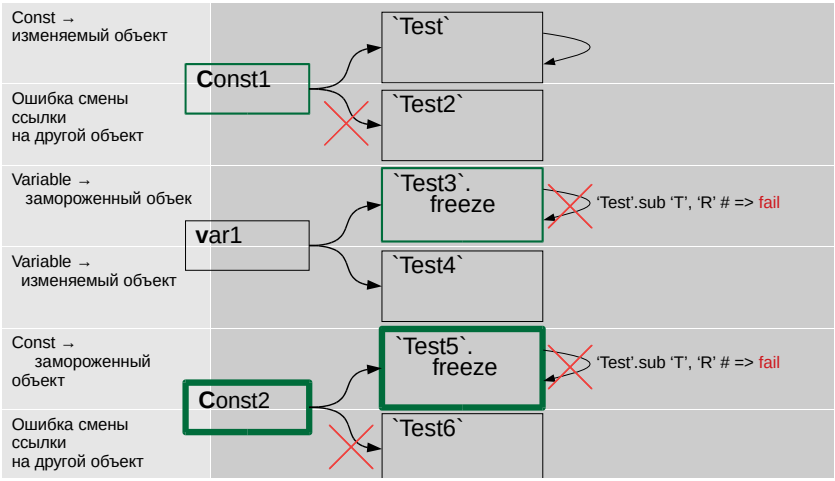

Рис. 3.1. Константы, переменные и замороженные объекты

Константа инициализируется только один раз и может ссылаться на модифицируемый объект. Изменение ссылки в константе невозможно. Но объект может измениться.

Переменная может получать ссылки на любые объекты. Если переменная ссылается на замороженный объект, он не изменится. Но переменная может получить новую ссылку на новый объект.

Единственный полный неизменяемый вариант константа, которая ссылается на замороженный объект. Любые попытки изменить ссылку у константы приведут к ошибке. Изменение объекта приведет к ошибке.

3.1.6. Область видимости атрибутов классов

Имеется ряд особенностей использования атрибутов экземпляров и классов.

Рассмотрим следующий пример:

```
@str = 'Hello!'
def test_print
  puts @str
end
test_print # => Hello!
```

Обратите внимание на то, что переменная @str не является глобальной, но доступна внутри метода ::test_print. Причина в том, что переменная @str, объявленная в начале кода, — это атрибут объекта (поскольку имеет префикс @). Переменная, к которой обращается метод ::test_print, также является атрибутом объекта. Сам же метод ::test_print объявлен вне класса, однако ввиду того, что в Ruby все существует в пределах классов, реально этот метод также присоединяется к классу Object. Поясним описание следующим примером:

```
@str = 'Hello!' # атрибут экземпляра
def test_print
  puts @str
  @@str2 = 'test' # атрибут уровня класса
end
test_print # проверяем одну переменную и выставляем другую

# метод для фильтрации вывода
def filter_output ar
   #выводим только известные нам имена
   filter = [ '@str', '@@str2', 'test_print', "#{__method__}"]
   ar.map {|x| x.to_s}.each do |text|
     puts "found #{text}" if filter.include? text
   end
end
# печатаем атрибуты и методы класса Object
puts "----------- Class Object ------------------"
filter_output Object.class_variables
filter_output Object.methods
filter_output Object.private_methods

# получаем все объекты-потомки Object
ObjectSpace.each_object(Object) do |o|
  # исключаем потомков и оставляем только экземпляры Object
  if o.instance_of? Object
    puts "----------- object ------------------"
    #p o
    # выводим атрибуты и методы экземпляра
    filter_output o.instance_variables +
```

```
                    o.methods +
                    o.private_methods
  end
end
```

Результат выполнения примера:

```
Hello!
----------- Class Object ------------------
found @@str2
found test_print
found filter_output
----------- object ------------------
found test_print
found filter_output
----------- object ------------------
found @str
found test_print
found filter_output
```

Из приведенного текста можем заключить, что объявленные методы доступны для Object и всех его потомков. Атрибут @str доступен только в пределах одного объекта Object. Атрибут @@str2 доступен для Object и всех его потомков.

3.2. Подключение файлов программы

Программы на языке Ruby могут состоять из нескольких файлов. Методами, которые подключают дополнительные файлы, являются #**require** (для системных библиотек) и #**require_relative** (для подключения файлов с указанием относительного пути). При этом все классы, модули, примеси и отдельные методы, объявленные в подключаемом файле, становятся доступными в файле, из которого вызван метод #require.

Особенность Ruby заключается в возможности декларировать методы класса или методы конкретного объекта в разных частях программы. Поэтому совершенно естественным является расширение классов по мере необходимости.

Ниже приведен пример подключения класса из другого файла.

Файл ext.rb:

```
class DistrClass
  def print_ext
     puts "Hi from #{__FILE__}!"
  end
end
```

Файл main.rb:

```
Require_relative 'ext.rb'
class DistrClass

  def print_int
     puts "Hi from #{__FILE__}!"
  end
end

ex = DistrClass.new
ex.print_ext # => Hi from ext.rb!
ex.print_int # => Hi from main.rb!
```

Отметим, что существует другой способ запуска кода программы из внешнего файла, который основан на том, что Ruby позволяет запустить любой код из строки с помощью метода ::eval на основании текста. Приведем пример.

Файл ext.rb:

```
def print_ext
   puts "Hi!"
end
puts 'This is external module'
```

Файл main.rb:

```
code = File.read 'ext.rb' # читаем код как файл и помещаем в строку
eval code
print_ext
```

Получим следующий результат:

```
This is external module
Hi!
```

Рассмотрим еще один способ запуска программ в изолированном окружении, изложенный в [7]. Глобальная переменная $SAFE позволяет ограничить возможности права доступа программы к различным ресурсам (обращение к переменным, ввод/вывод и пр.). Метод #load имеет второй параметр wrap, который сообщает о том, что программа должна быть загружена в изолированном окружении. Таким образом обеспечивается возможность безопасно запустить непроверенный код:

```
Thread.start do
  $SAFE = 4
  load('ext.rb', true)
end
```

3.3. Модули и примеси

В отличие от C++ или Java язык Ruby позволяет описывать тело класса в нескольких файлах. Также ввиду динамической природы языка Ruby доступна подмена методов объектов на этапе выполнения. Эти возможности широко используются для создания так называемых патчей библиотек, т. е. заплаток, устраняющих определенные выявленные проблемы. Однако подход к написанию кода в таком стиле называется термином monkey-patching, поскольку легко приводит к возникновению кода, состоящего из заплаток, который проанализировать в целом невозможно. Поэтому подмену методов существующих классов и объектов следует делать весьма осторожно. Тем не менее этот подход применяется довольно часто и особенно широко, например, в случае преобразования программных интерфейсов Java в специфический для Ruby вид, если используется jRuby.

Модули синтаксически похожи на классы, однако в отличие от классов, не могут создать экземпляр модуля. Модули определяются как коллекции констант и методов. Основное назначение модулей заключается в выделении некоторых общих свойств, которые могут быть использованы в нескольких классах (как примеси). В этом контексте механизм модулей для Ruby является заменой отсутствующего множественного наследования. Другое назначение модулей — ограничение пространства имен с тем, чтобы имена методов и констант не пересекались с именами в других файлах/библиотеках.

Примеси — механизм, позволяющий включать модули в различные классы. Пример (см. http://rails.vsevteme.ru/2009/02/28/samorazvitie/5-metaprogramming-patterns-18-kyu-primesi):

```
module Sum
  def sum
    inject {|s, element| s + element }
  end
end
class Array
  include Sum
end
[1,2,3,4,5].sum #=> 15
```

В стандартный класс Array добавлен метод #sum, который взят из модуля Sum. Здесь следует обратить внимание на то, что

подключенный модуль Sum будет влиять только на созданные **после** этого момента массивы, но не на массивы, созданные ранее!

Приведем другой пример. Вместо добавления примеси единственному классу добавим новый метод в стандартный модуль Enumerable, который уже подключен как примесь большинству стандартных классов:

```
require 'set'
# расширяем библиотечный модуль Enumerable
module Enumerable
  def sum
    inject {|m, element| m + element }
  end
end
# и тогда мы получаем sum для всех классов контейнеров,
# но только для объектов, созданных после этого!
Set[5,3,1].sum       # => 9
puts(('a'..'z').sum) # => 'abcdefghijklmnopqrstuvwxyz'
{1=>'a',2=>'b'}.sum  # => [1, "a", 2, "b"]
```

Контрольные вопросы и задания

1. Что представляет собой класс для языка Ruby?

2. В чем различие идентификаторов @a и @@b? Приведите конкретный пример использования.

3. Какими способами можно проверить, является ли класс Class1 предком Class2 и как можно выявить, есть ли у них общие предки (за исключением класса Object)?

4. Напишите программу, отвечающую на вопрос 3. Для проверки программу сохраните в rb-файле и запустите с использованием Ruby.

5. Изложите основные способы подключения дополнительных файлов программ для Ruby.

6. Напишите программу, которая позволяет ввести с клавиатуры произвольную строку кода на языке Ruby, после чего запускает ее на выполнение.

7. Могут ли классы иметь тело, распределенное по нескольким файлам?

8. Каково назначение модулей и в чем их отличие от классов?

9. Что такое примеси?

10. Приведите пример добавления однократно написанного метода в два класса: Class1 и Class2, которые не находятся в отношении предок-потомок.

4. КОМПАКТНЫЙ КОД

Одной из причин популярности языка Ruby является то, что код программы может быть реализован очень коротко, но при этом сохранит понятность и выразительность. Этот раздел посвящен аспектам, позволяющим реализовать такой код.

4.1. Блоки

Ключевым элементом Ruby, позволяющим скомпоновать код, является блок. Общая информация по блокам уже приводилась ранее, поэтому приведем несколько конкретных примеров их использования.

Рассмотрим метод поиска элемента массива по заданному условию. Следует заметить, что поиск элемента в произвольном массиве предполагает проход по всем элементам этого массива. Изменяется только условие, которое может быть достаточно сложным [7]:

```
class Array
  # универсальный метод поиска
  def find
    for i in 0...size
      # получаем текущий элемент массива
      value = self[i]
      # вызываем код блока для проверки условия. Выходим если true
      return value if yield(value)
    end
    # ничего не нашли. Выходим.
    return nil
  end
end
# задаем массив и сложное условие поиска v*v > 30
[1, 3, 5, 7, 9].find {|v| v*v > 30 } # => 7
```

Рассмотрим пример по организации блока для безопасной работы с файлами [7]:

```
class File
  # метод, обеспечивающий открытие и закрытие файла
```

```
  def self.open_and_process(*args)
    f = File.open(*args)
    # вызываем код из блока
    yield f
    f.close()
  end
end
File.open_and_process("testfile", "r") do |file|
  # выполняем обработку файла, не заботясь о его открытии/закрытии
  while line = file.gets
    puts line
  end
end
```

Этот пример приведем лишь для иллюстрации того, как может быть организована обработка файла. Однако библиотечный метод File#open имеет две возможные формы вызова: с блоком и без него. В случае вызова без блока File::open открывает файл и возвращает объект класса File. Возможность определить, был ли вызов осуществлен с блоком или нет, реализуется методом #block_given?.

Пример реализации метода [7]:

```
class File
  def self.my_open(*args)
    result = file = File.new(*args)
    # Если блок есть, вызвать его код и закрыть файл.
    if block_given?
      result = yield file
      file.close
    end
  # result содержит либо объект File, либо то, что вернул блок
  return result
  end
end
```

Рассмотрим практический пример. Предположим, что необходимо обеспечить формирование некоторых файлов по шаблону. Причем количество сгенерированных файлов зависит от входных данных. Существует много задач, для каждой из которых имеются свои шаблоны. В данном случае необходимо написать программу, понимающую некоторый набор данных и обрабатывающую шаблоны. Очевидно, что для каждого типа шаблона состав данных для замены будет различаться. Поскольку для разных задач обработка шаблонов в целом совпадает, целесообразно написать программу так, чтобы код, ответственный за обработку шаблона конкретного типа, не содержал общей части обработки.

Для простоты будем считать, что необходимо сгенерировать набор почтовых сообщений для пользователей, перечисленных в файле, причем шаблоны находятся в директории mail_template в двух файлах — template.xml и message_headers.properties.

В качестве входных данных используем файл mail_template_data:

```
# общая часть
SMTP_HOST='192.168.1.1'
SMTP_PORT='25'
MAIL_FROM = 'control_system@test'

# список имен пользователей, их адресов и темы сообщений
#   id        mail_to        subject
DATA=[
  ['Ivanov', 'ivanov@test', 'Emergency !'],
  ['Petrov', 'petrov@test', 'Emergency !'],
]
```

Программу для обработки шаблонов именно этого типа реализуем следующим образом (файла generator1.rb):

```
#!/usr/bin/env ruby
# подключаем файл с логикой обработки
require_relative 'include/generator_common.rb'
#объявляем константы источника шаблонов и директорией назначения
SRC_DIR="mail_template"
DEST_PREF="result/"

# запускаем на обработку,

#    причем данные берем из файла "#{SRC_DIR}_data"
process_data "#{SRC_DIR}_data" do |i, name, mail_to, subject|
  id = "#{i}: #{name}"
  # заменяем в файле 'template.xml' символы,
  #окруженные знаками %...% на значения из таблицы
  replace 'template.xml', {
          '%SMTP_HOST%' => SMTP_HOST,
          '%SMTP_PORT%' => SMTP_PORT,
          '%MAIL_FROM%'=> MAIL_FROM,
          '%MAIL_TO%' => mail_to,
          '%MAIL_SUBJECT%' => subject,
          '%id%' => id,
          }

  # аналогично заменяем в файле  message_headers.properties

  replace 'message_headers.properties',
          '%MAIL_FROM%' => MAIL_FROM

  # возвращаем суффикс для дописывания к имени файла результата
  "_#{name}"
end
```

Обратите внимание на метод process_data. Этот метод получает имя файла, а передаваемые параметры в блок соответствуют

колонкам массива DATA, который описан в файле mail_template_data. Метод #replace в коде программы предназначен для замены в файлах шаблонов на указанные значения.

Рассмотрим файл include/generator_common.rb, который скрывает общую часть обработки, не зависящую от конкретного шаблона:

```
require 'fileutils'

# замена значений в файле. from_to - хеш пар замены

def replace filename, from_to
src = "#{SRC_DIR}/#{filename}"
  puts text = File.read(src)
  # для каждой пары замены меняем текст
  from_to.each {|from, to| text.gsub!(from, to) }
  # записываем результат обработки
  File.open('tmp/'+filename, "w") { |file| file << text }
end

def process_data data_fn

  # загружаем файл с данными
  eval File.read data_fn

  # проходим по массиву DATA из файла

  DATA.each_with_index do |item, i|
    # элементы массива DATA являются массивами.
    # Раскрываем вложенные массивы операцией *item
    # и передаем их в блок
    suff = yield (i+1).to_s, *item

    # копируем все сформированные шаблоны

    # в результирующую директорию
    Dir.glob 'tmp/*' do |fn|
      FileUtils.copy
        fn,
        DEST_PREF + File.basename(fn) +
        ((suff.is_a? String)&&(!suff.empty?) ? suff : (i+1).to_s)
    end
  end
end
```

Этот пример иллюстрирует возможности использования блока с переменным числом параметров, что позволяет в дружественной для программиста форме реализовывать код программы.

Напомним, что блок, по сути, является объектом типа Proc. Язык Ruby позволяет создать фрагмент кода как объект типа Proc, выполнение которого можно осуществить в любой момент времени.

Пример:

```
a = Proc.new {|name| puts "Hello, #{name}!"}
a.call 'world'
```

Обратите внимание на то, что код объекта Proc может вызвать побочные эффекты для вызывающей его программы. В следующем примере никогда не будет напечатана строка «proc_test end»:

```
def proc_test
  Proc.new {puts 'proc'; return}.call
  puts 'proc_test end' # сюда не попадаем никогда!
end
proc_test
```

Причина заключается в том, что метод return будет вызван внутри метода proc_test, а не внутри Proc. Для того чтобы избежать указанного поведения программы, необходимо использовать конструкцию lambda, которая представляет собой не просто участок кода для подстановки, а полностью автономный объект-функцию.

4.2. Перечислители

Язык Ruby имеет два типа перечислителей: класс Enumerator и примесь Enumerable.

Класс Enumerator предназначен прежде всего для обеспечения возможности пошагового прохода по элементам структуры данных. Основными методами для него являются #next, #next_values, #peek, #peek_values, #rewind. Полный список методов приведен по адресу https://ruby-doc.org/core-2.7.0/Enumerator.html

Основные способы применения класса Enumerator можно проиллюстрировать следующими примерами из документации:

```
e = [1,2,3].each   # returns an enumerator object.
puts e.next   # => 1
puts e.next   # => 2
puts e.next   # => 3
puts e.next   # raises StopIteration
```

```
a = [1,2,3]
e = a.to_enum
p e.next   #=> 1
p e.next   #=> 2
p e.next   #=> 3
p e.next   #raises StopIteration
```

В то же время существует примесь Enumerable, которая включена в такие классы, как String, Array, Hash, Range и даже в Enumerator. (Более подробно об Enumerator см. на сайте https://ruby-doc.org/core-2.7.0/Enumerator.html).

Эта примесь обеспечивает реализацию следующих методов: #all? #any? #chunk #collect #collect_concat #count #cycle #detect #drop #drop_while #each_cons #each_entry #each_slice #each_with_index #each_with_object #entries #find #find_all #find_index #first #flat_map #grep #group_by #include? #inject #lazy #map #max #max_by #member? #min #min_by #minmax #minmax_by #none? #one? #partition #reduce #reject #reverse_each #select #slice_before #sort #sort_by #take #take_while #to_a #zip.

Все классы, включающие данную примесь, получают возможность использования этих методов. Заметим, что метод #each классы реализуют самостоятельно в соответствии с собственной структурой данных. Большинство этих методов реализуют циклы обработки элементов структуры данных, причем условие или выполняемое для каждого элемента действие реализуется в блоке этого метода. Например, имеется возможность записать как obj.any?, так и obj.any?**{|x| x>0}**. Подобные методы позволяют полностью отказаться от использования традиционных циклов **while, for, until**.

Рассмотрим простой пример работы со строками:

```
# coding: utf-8

str = "Некоторая строка, содержащая символы и слова"
puts str
# проверим, есть ли в строке символ ','
flag = str.chars.any? {|x| x == ','}

puts "В строке имеется символ \',\'" if flag
print str.chars.sort.join # отсортируем символы перед выводом
```

Результат выполнения программы:

```
Некоторая строка, содержащая символы и слова
В строке имеется символ ','
     ,Наааааввдеежиикклллмоооооорррсссттщыяя
```

В приведенном примере использован метод (квантор существования, если руководствоваться его математическим названием) #any?. Он может быть применен к одному из представлений строки: к lines — линиям (т. е. фрагментам,

разбитым указанным символом), к chars — отдельным символам или к bytes — байтам. Задача этого квантора выявить существование условия, поэтому при формировании отладочного вывода строка обрывается на первом найденном элементе, т. е. как только сравнение x == ',' вернет true. Второй момент, на который следует обратить внимание, — это сортировка строки с помощью метода #sort. Аналогично #any? этот метод применен не к объекту строки, а к массиву символов.
В данном случае использованы сравнение по умолчанию и метод join для склеивания отдельных символов в строку.

4.3. Именование методов

При написании программ на Ruby очень важно обеспечить читаемость кода программы. Поэтому при выборе имен методов следует придерживаться имен, соответствующих смыслу выполняемых программой действий. Причем в стандартных библиотеках Ruby существуют методы с разными именами, но выполняющие одно и то же действие.

Рассмотрим несколько примеров. Для начала перечислим несколько идентичных или почти идентичных методов примеси Enumerable:

- #map, #collect;
- #find_all, #select, #reject(обратное к #select);
- #reduce, #inject;
- #find, #detect;
- #include?, #any? (значение в блоке).

Логические условия необходимо писать так, чтобы код программы максимально соответствовал решаемой задаче. Следует избегать усложнения условий за счет логических отрицаний, а также по возможности логического отрицания как такового.

Например, вместо кода:

```
puts a if !c.find (a)
```

лучше написать

```
puts a unless c.find (a)
```

или более близкий по смыслу вариант кода

```
puts a unless c.member? a
```

Проверим ненулевую длину массива:

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

Вместо кода

```
a.size != 0  #=> true
```

лучше написать код

```
a.length > 0 #=> true
a.any?       #=> true
```

Для проверки нулевой длины лучше использовать следующий метод:

```
[].empty?    #=> true
```

4.4. Динамические методы

Поскольку Ruby является динамическим языком, в котором имеется возможность присоединить новый метод к объекту или классу, в нем также предусмотрены методы, позволяющие контролировать обращение к несуществующим методам класса. Для этого используется метод #method_missing, который может быть определен для конкретного класса или неявно для класса Object.

Рассмотрим следующий пример:

```
class String
  [:red, :green, :blue].each do |m|
    define_method("#{m}") do "<span style='color: #{m}'>#{self}</span>"}
  end
end
puts "test".green, "test".red, "test".blue
```

В этом примере использован метод define_method, который по массиву символов динамически создает соответствующие методы для класса String. В результате выполнения данного кода получим

```
<span style='color: green'>test</span>
<span style='color: red'>test</span>
<span style='color: blue'>test</span>
```

Приведем пример с методом #method_missing. Определим ряд констант, эквивалентных именам вызываемых методов, и поставим им в соответствие действия, которые должны быть выполнены при вызове одноименного метода:

```
# конструируем Hash в форме symbol => lambda
# Выражение ->(var){ … } представляет собой код,
#    который должен быть выполнен
Elements = {
  html:  ->(var){ "<html>#{var}</html>\n" },
  head:  ->(var){ "<head>#{var}</head>\n" },
  title: ->(var){"<title>#{var}</title>\n" },
  body:  ->(var){ "<body>#{var}</body>\n" },
  div:   ->(var){ "<div>#{var}</div>\n" },
  br:    ->{ "<br/>" }
}
# перехватываем любое упоминание неизвестного метода
def method_missing(meth, *args, &block)
   (func = Elements[meth]) ?
      (block ? func.call( block.call ) : func.call) :
      # если не знаем, что это, вызываем обработчик предка
      super.method_missing(meth, *args, &block)
end
# пишем код с использованием неизвестных методов
result = html do
  head { title { "Test page" } } +
  body do
    div { "Uncolored string" + br + "test string".green }
  end
end
puts result
```

В консоли получим следующий результат:

```
<html><head><title>Test page</title>
</head>
<body><div>Uncolored  string<br/><span  style='color:  green'>test
string</span></div>
</body>
</html>
```

Отметим, что несмотря на возможность перехватить обращение к несуществующему методу, целесообразно динамически определять необходимые методы с помощью Module#define_method, поскольку вызов метода method_missing, по сути, является аварийной ситуацией и требует значительно больше времени, чем вызов уже созданного метода.

4.5. Элементы функционального программирования в Ruby

Императивное программирование рассматривает выполнение программы как последовательность изменения состояний некоторых переменных на основании строго заданной последовательности действий. Для функционального

программирования свойственно описывать вычислительный процесс как непрерывный процесс вычисления некоторых функций в математическом смысле, а не в смысле функций процедурного программирования. При этом аргументами функций могут быть и другие функции, но нет переменных, которые хранят какой-то конкретный результат. Основой функционального программирования служит lambda-исчисление. Ruby не является функциональным языком программирования, но предоставляет возможность писать программы в функциональном стиле.

Для функционального языка программирования необходимо, чтобы объектом вычисления была функция. Ruby имеет класс Proc и конструкцию lambda, результатом которой являются объект-функции. Рассмотрим примеры создания lambda-объектов с помощью двух равнозначных способов:

```
p a = ->(x) {x*x}
p a.call 2 # получим 4

p b = lambda{|x| x+x}

p b.call 3 # получим 6
```

Принципиальное отличие lambda от объекта, созданного как Proc.new, заключается в том, что lambda является полностью изолированным объектом и может только вернуть значение, в то время как Proc, по сути, осуществляет подстановку кода. Например, в результате выполнения кода

```
def lambda_test
  ->{puts 'lambda'; return}.call
  puts 'lambda_test end'
end
lambda_test
```

получим

```
lambda
lambda_test end
```

Созданные lambda-объекты могут быть переданы в качестве аргументов другим функциям или использованы в любой момент времени для выполнения соответствующего им кода. Это позволяет реализовать функции высших порядков. Например, следующий код обеспечит вычисление квадратов элементов массива:

```
a = ->(x){ x*x }
p (1..10).map(&a) # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Для функционального языка существуют требования неизменности полученных значений, а также требование отсутствия побочных эффектов, т. е. никакое полученное значение не может быть изменено. Применение любой функции к объекту приводит к созданию нового объекта. Кроме того, никакая функция не может зависеть от значений вне ее аргументов. Это гарантирует, что вызов функции с одними и теми же аргументами всегда приведет к получению одного и того же результата.

Ruby не имеет средств, которые гарантировали бы выполнение указанных требований на этапе написания кода, но эти требования могут быть реализованы организационными мерами, например соглашением о кодировании.

Для реализации функционального стиля необходимо, чтобы любой метод возвращал некоторое значение (для Ruby — объект). Это позволяет записывать компактные цепочки вызовов. Например, запись

```
(1..10).map{|x| x*x}.reduce(:+)
```

представляет собой последовательный процесс преобразования данных, в котором на каждом шаге создается новый объект. То же может быть записано следующим образом:

```
a = (1..10)          # => 1..10 (Range)
# создаем новый массив, где каждый элемент преобразуем
#    по заданному коду
b = a.map{|x| x*x}  # => [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
# сворачиваем массив, применив ко всем элементам метод с именем '+'
c = b.reduce(:+)     # => 385
```

Пример с большим количеством кода, но не более сложный:

```
%w[ 1 2 3 4 5].map(&:to_i).map{|x| [x*x, x]}.
   inject({}){|res, item| res.merge!(item[1] => item[0])}
```

Эту запись также можем разбить на более привычную для императивного языка программирования последовательность:

```
# формируем массив строк
p a = %w[ 1 2 3 4 5]       # => ["1", "2", "3", "4", "5"]
# Каждый элемент массива преобразуем в число
p b = a.map(&:to_i)        # => [1, 2, 3, 4, 5]
# Каждый элемент массива преобразуем в массив
#  из двух чисел [квадрат, число]
p c = b.map{|x| [x*x, x]}
```

```
#     => [[1, 1], [4, 2], [9, 3], [16, 4], [25, 5]]

# создаем новый объект хеш - {}
#   и наполняем его парами ключ-значение, которые берем
#   из элементов массива.
d = c.inject({}) do |res, item|
   res.merge!(item[1] => item[0])
end # => {1=>1, 2=>4, 3=>9, 4=>16, 5=>25}
```

Обратите внимание на то, что метод map предназначен для отображения каждого элемента массива в элемент нового массива, причем количество элементов будет абсолютно идентично исходному массиву. В случае reduce создан единственный новый объект, который собрал значения исходного массива с общим результатом.

Несложно убедиться, что даже если отбросить комментарии, код в императивном стиле будет занимать существенно больше места, чем код в виде последовательной цепочки вызовов, но для этого каждый метод должен возвращать объект, над которым можно продолжить процесс преобразования.

Метод **map** имеет синоним **collect**, причем оба они создают новые объекты. При необходимости модифицировать существующий объект-массив, а не создавать новый, следует использовать метод **map!** или **collect!**.

Методы inspect и его синоним reduce позволяют вставить объект-аккумулятор, который накапливает результат выполняемого преобразования данных.

Напомним, что при реализации метода с блоком единственный способ получить значение — это проанализировать возвращаемое методом yield значение. Попробуем представить реализацию метода inject в виде простого примера. Присоединим наш метод our_inject к классу Array:

```
class Array
  # добавляем новый метод массиву
  # аргументом является объект-аккумулятор с начальным значением
  def our_inject(init_val)
    # заводим внутренний аккумулятор.
    acc = init_val
    # проходим по всем элементам массива
    each do |item|
      # для каждого из них вызываем блок, которому передаем
      # текущее значение аккумулятора и текущий элемент массива.
      # Обратите внимание — передаем массивом на два элемента
      # Запоминаем в аккумуляторе значение, возвращенное из блока
      acc = yield [acc, item]
```

```
    end
    acc
  end
end
# вызываем метод и присоединяем к нему блок,
# в котором проводим вычисления
p [1, 2, 3, 4].our_inject(50.0) {|acc, x| acc/x}
#     => 2.0833333333333335
```

Этот пример иллюстрирует, что в качестве аккумулятора может быть передан любой объект, а логику работы определяем сами в переданном блоке.

Существуют возможности передавать операции в более компактной форме. Например, очень часто используется преобразование строки — число для элементов массива. Это может быть записано в форме:

```
%w[ 1 2 3 4 5].map {|str| str.to_i}
```

или гораздо короче путем передачи в качестве блока ссылки на существующий у объектов массива метод:

```
%w[ 1 2 3 4 5].map(&:to_i)
```

То же самое можно выполнить в отношении методов inject и reduce. Они могут быть использованы как с блоком, так и с указанием метода, который необходимо применить. Например, для перемножения элементов массива достаточно написать

```
# 1 — начальное значение аккумулятора, :* - имя метода
[1,2,3,4,5].reduce(1, :*) # => 120
```

Используем ряд примеров из проекта, созданного для иллюстрации функционального стиля Ruby (см. https://github.com/tokland/tokland/wiki/RubyFunctionalProgramming).

Некоторые требования функционального программирования приведены в табл. 4.1.

Таблица 4.1

Некоторые требования функционального программирования

Неправильно	Правильно (в функциональном стиле)

`indexes = [1, 2, 3]`
`# меняем массив`
`indexes << 4`
`indexes # [1, 2, 3, 4]`
	`indexes = [1, 2, 3]`
	`# создаем новый массив`
	`all_indexes = indexes + [4] # [1, 2, 3, 4]`

`hash = {:a => 1, :b => 2}`
`# меняем объект hash`
`hash[:c] = 3`
`hash`
	`hash = {:a => 1, :b => 2}`
	`# создаем новый объект hash`
	`new_hash = hash.merge(:c => 3)`

`string = "hello"`
`# меняем строку`
`string.gsub!(/l/, 'z')`
`string # "hezzo"`
	`string = "hello"`
	`#создаем новую строку`
	`new_string = string.gsub(/l/, 'z') # "hezzo"`

`output = []`
`output << 1`
`output << 2 if i_have_to_add_two`
`output << 3`
	`# создаем массив, затем убираем пустые`
	`output =`
	`  [1, (2 if i_have_to_add_two), 3].compact`

`# переопределяем переменную`
`number = gets`
`number = number.to_i`
	`# новый тип значения – новая переменная`
	`number_string = gets`
	`number = number_string.to_i`

Не всегда следует слепо выполнять эти требования. Функциональные языки программирования не вычисляют значения сразу. Поэтому операция добавления элемента в массив для них останется лишь командой на добавление, которая будет когда-то выполнена, но не приведет к созданию копии массива. Для Ruby хранение переменной, указывающей и на старый массив, и на новый, приведет к тому, что в памяти процесса будут храниться оба массива. Поэтому следует придерживаться функционального стиля, но оценивать целесообразность этого с учетом ограниченности ресурсов вычислительной системы.

Рассмотрим также типовые конструкции Ruby, которые могут быть свернуты в функциональном стиле.

1. Конструкция **пустой массив + each + вставка значения = = map**.

Неправильно:

```
dogs = []
["milu", "rantanplan"].each do |name|
  dogs << name.upcase
end
```

```
dogs # => ["MILU", "RANTANPLAN"]
```

Правильно:

```
dogs = ["milu", "rantanplan"].map do |name|
  name.upcase
end # => ["MILU", "RANTANPLAN"]
```

2. Конструкция **пустой массив + each + условное добавление -> -> select/reject**.

Неправильно:

```
dogs = []
["milu", "rantanplan"].each do |name|
  if name.size == 4
    dogs << name
  end
end
dogs # => ["milu"]
```

Правильно:

```
dogs = ["milu", "rantanplan"].select do |name|
  name.size == 4
end # => ["milu"]
```

3. Конструкция **инициализация + each + накопление результата -> inject**.

Неправильно:

```
length = 0
["milu", "rantanplan"].each do |dog_name|
  length += dog_name.length
end
length # => 14
```

Правильно (accumulator обеспечивает накопление):

```
length = ["milu", "rantanplan"].inject(0) do |accumulator, dog_name|
  accumulator + dog_name.length
end # => 14
```

В случаях когда между аккумулятором и аргументом выполняется простая операция, можно не использовать блок, а передать название бинарной операции в виде объекта-символа:

```
length = ["milu", "rantanplan"].map(&:length).inject(0, :+) # 14
```

4. Конструкция **инициализация + присвоение по условию + + присвоение по условию + ...**

Рассмотрим следующий код:

```
name = obj1.name
name = obj2.name if !name
name = ask_name if !name
```

В функциональном стиле этот код может быть записан в более короткой форме, поскольку любое значение, отличное от false и nil, вызовет единственное присвоение:

```
name = obj1.name || obj2.name || ask_name
```

Приведем более сложный пример кода:

```
def get_best_object(obj1, obj2, obj3)
  return obj1 if obj1.price < 20
  return obj2 if obj2.quality > 3
  obj3
end
```

Этот код может быть записан в гораздо более прозрачной форме:

```
def get_best_object(obj1, obj2, obj3)
  if obj1.price < 20
    obj1
  elsif obj2.quality > 3
    obj2
  else
    obj3
  end
end
```

Следует помнить, что для Ruby существует результат выполнения для любой операции. Это может быть использовано, например, в следующем случае:

```
if found_dog == our_dog
  name = found_dog.name
  message = "We found our dog #{name}!"
else
  message = "No luck"
end
```

В компактной форме этот же код может быть записан так:

```
message =
  (if found_dog == my_dog
     name = found_dog.name
     "We found our dog #{name}!"
   else
     "No luck"
   end)
```

Подведем краткий итог по элементам функционального программирования. На рис. 4.1-4.3 показано использование #map,

#select и #inject, соответственно. Все эти методы имеют внутренний цикл, проходящий по всем элементам. Функция обработки данных задается внутри блока, передаваемого этим методам.

```
[1, 2, 3, 4, 5].map { |x| x ** 2 }
  # => [1, 4, 9, 16, 25]
```

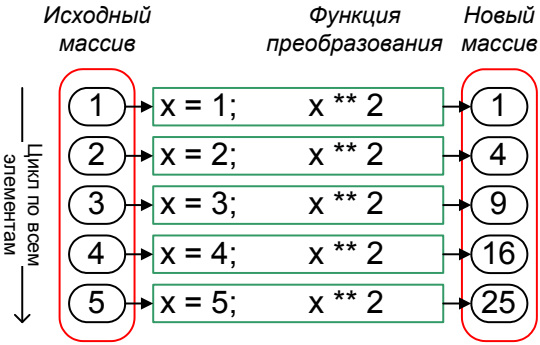

Рис. 4.1. Принцип использования метода map

```
[1, 2, 3, 4, 5].select { |x| x.even? }
  # => [2, 4]
```

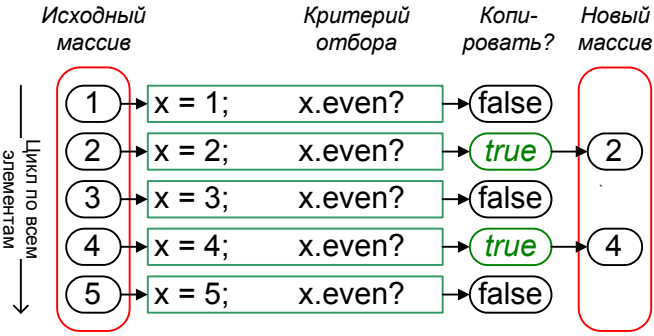

Рис. 4.2. Принцип использования метода select

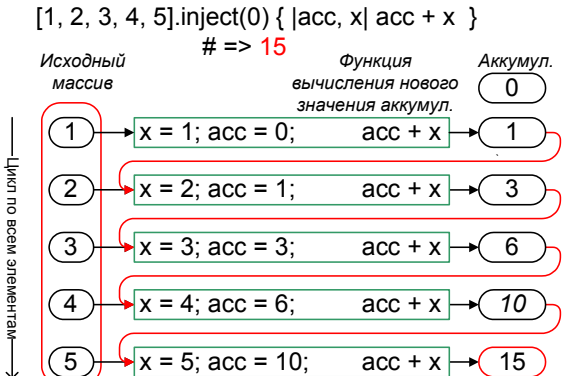

Рис. 4.3. Принцип использования метода inject

На рисунке 4.4 иллюстрируется интерпретация длинной цепочки вызовов. Вызов любого метода порождает новый объект. Поскольку язык имеет динамическое освобождение памяти, нет необходимости создавать промежуточные переменные, хотя это возможно. Соответственно, можем рассматривать цепочку вызовов как последовательные вызовы методов соответствующих объектов, порожденных вызовом метода предшествующего в цепочке объекта.

```
'1 2 3 4 5'.split.map(&:to_i).select(&:even?).map {|x| x**2}.reduce(&:+)
```

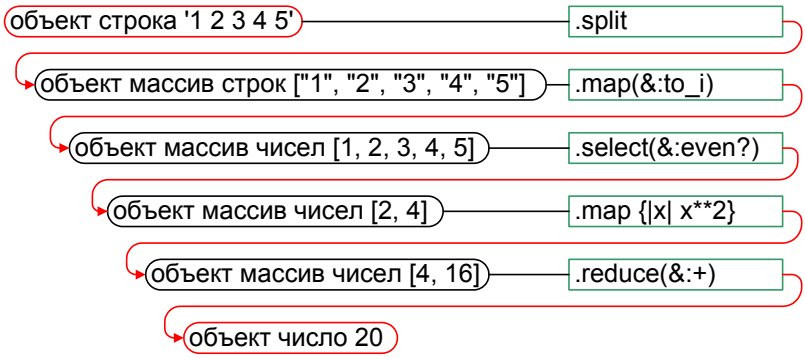

Рис. 4.4. Цепочка вызовов и ее интерпретация.

4.6. Отложенные вычисления

Как уже упоминалось ранее, функциональные языки программирования в отличие от императивных не вычисляют результат функции сразу. Результат будет вычислен только в том случае, если он необходим для вычисления следующей функции. Причина этого заключается в том, что последовательное применение функций может привести к упрощению выражения (именно так, как это происходит при упрощении математических выражений), а это сделает ненужным вычисления и позволит избежать потери точности на ограниченных машинных регистрах. Вычисление функций подобным образом называется отложенным вычислением, или Lazy lambda-calculus.

Дополнительный эффект отложенных вычислений — возможность распараллелить процесс вычисления без какого-либо вмешательства программиста.

Язык Ruby не имеет средств для выполнения отложенных вычислений в функциональном смысле, но в Ruby версии 2.0 появился класс Enumerator::Lazy.

Класс Enumerator::Lazy предназначен для реализации концепции отложенных вычислений не на функциях как таковых, а на последовательностях. Основная идея отложенных вычислений заключается в том, что функция, на которую ссылается переменная, будет выполнена лишь тогда, когда потребуется результат, и лишь на то количество значений последовательности, которое необходимо. Для этого предусмотрены соответствующие методы — #take, #first.

Рассмотрим простой пример: необходимо найти сумму 10 первых натуральных чисел, квадрат которых кратен 5. Решение в императивном стиле выглядит следующим образом:

```
n, num_elements, sum = 1, 0, 0
while num_elements < 10
if n**2 % 5 == 0
    sum += n
    num_elements += 1
  end
  n += 1
end
p sum #=> 275
```

Решение в функциональном стиле с отложенными вычислениями имеет следующий вид:

```
p (1..1.0/0).lazy.select { |x| x**2 % 5 == 0 }.take(10).inject(:+)
  #  => 275
```

Обратите внимание на то, что множество натуральных чисел задано через диапазон и операцию 1.0/0, которая приведет к получению значения Float::INFINITY или Infinity. Попытка целочисленного деления 1/0 приведет к исключению.

Для того чтобы понять разницу между обычными цепочками вызовов и lazy следует вставить отладочную печать. Сравним результаты вывода для lazy и традиционной цепочки вызовов.

Для lazy:

```
(1..1.0/0).lazy.select{|x| puts %Q[select from: #{x}]; x.even?}.
           map{|x| puts %Q[map #{res=x*x}]; res}.
           first(3)
```

```
select from: 1
select from: 2
map 4
select from: 3
select from: 4
map 16
select from: 5
select from: 6
map 36
=> [4, 16, 36]
```

Для традиционной цепочки вызовов:

```
(1..10).select{|x| puts %Q[select from: #{x}]; x.even?}.
           map{|x| puts %Q[map #{res=x*x}]; res}.
           first(3)
```

```
select from: 1
select from: 2
select from: 3
select from: 4
select from: 5
select from: 6
select from: 7
select from: 8
select from: 9
select from: 10
map 4
map 16
map 36
map 64
map 100
=> [4, 16, 36]
```

Легко заметить различие в том, что для lazy вызовы select происходили только после того, как предыдущее значение было обработано в последующем методе map. В традиционном же варианте все значения исходного массива были обработаны сразу.

Подробнее про отложенные вычисления можно прочитать по следующим ссылкам:

https://ruby-doc.org/core-2.7.0/Enumerator/Lazy.html

http://patshaughnessy.net/2013/4/3/ruby-2-0-works-hard-so-you-can-be-lazy

Контрольные вопросы и задания

1. Что будет выведено на экран:

```
a, b = 1, 2
c = a + b = a - b
p a,b,c
```
?

Для проверки используйте irb.

2. Что такое блок языка Ruby?

3. Приведите пример кода, в котором реализуется метод для массива, позволяющий блоку накапливать значения.

4. Приведите пример вызова методов, к которым будет присоединен блок, Proc, Proc::lambda. Поясните различия.

5. В чем различие объектов класса Proc и lambda (помимо синтаксического различия)?

6. Приведите пример кода для динамического создания метода, имеющего имя, введенное с клавиатуры. Для проверки программу сохранить в rb-файл и запустить с использованием Ruby.

7. В чем смысл методов map и reduce? Приведите пример кода, когда использование этих методов целесообразно (исходный вариант и результат после применения map и reduce).

8. Перепишите с помощью select:

```
result = []
arr.each do |name|
   if name.lenght > 5
     result << name
   end
end
```

Для проверки программу сохраните в rb-файле и запустите с использованием Ruby.

9. Разверните выражение в процедурный стиль

```
(1..3).select{|x| x.odd?}.inject(:+)
```

Для проверки программу сохраните в rb-файле и запустите с использованием Ruby.

10. В чем отличие lazy-вычислений от последовательного преобразования объектов?

5. ТЕСТИРОВАНИЕ И ОТЛАДКА

Существенным этапом при создании современных программных продуктов является тестирование. Язык Ruby позволяет реализовать тесты для автоматической проверки кода на всех этапах написания программ. Рассмотрим подходы Test::Unit и Rspec.

Для написания встроенных unit-тестов существует стандартная библиотека Test::Unit. Приведем пример. Программа создает двоичное дерево и обеспечивает вывод значений в различном порядке:

```
class Tree
attr_accessor :left, :right, :data
protected :left, :right, :data

  def initialize(x=nil)

    @left, @right = nil, nil
    @data = x
  end

  def insert(x)

    if @data == nil
      @data = x
    elsif x <= @data
      if @left == nil
        @left = Tree.new(x)
      else
        @left.insert(x)
      end
    else
      if @right == nil
        @right = Tree.new(x)
      else
        @right.insert(x)
      end
    end
  end

  def inorder()
    @left.inorder{|y| yield y} if @left != nil
```

```
    yield @data
    @right.inorder{|y| yield y} if @right != nil
  end
  def inbackorder()
    @right.inbackorder{|y| yield y} if @right != nil
    yield @data
    @left.inbackorder{|y| yield y} if @left != nil
  end
end
```

Протестируем поведение этой программы следующим образом:

```
require "./tree.rb"

# items = 100.times.map{ Random.rand(200) - 100 }
items = [35, 1, 24, 2, -4, 3, 25, 4, 94, 5, 0, 6, 14, 7]

tree = Tree.new
items.each{ |x| tree.insert(x) }

#обратите внимание на эквивалентность здесь print "\n" и puts
tree.inorder{|x| print "#{x} "}; print "\n"
puts tree.inbackorder{|x| print "#{x} "}
```

Результат этого вывода ничего не говорит о том, правильно ли работает программа. С использованием Minitest::Unit можно реализовать конкретные тесты. Для того чтобы создать программу тестирования, необходимо создать класс, являющийся потомком Minitest::Unit::TestCase. При этом будут выполнены все методы, которые определены в этом классе и имеют префикс test_. Каждый из них является самостоятельным тестом. Причем порядок их выполнения — алфавитный. Специальный метод **setup** будет вызваться **перед** каждым тестом и может содержать инициализацию всех необходимых данных. Проверку правильности выполнения следует проводить через специальные утверждения **assert_**, например ::assert_equal. Или утверждения о том, что выражение ложно, имя которых начинается с префикса **refute_**. Тест должен иметь по крайней мере одну проверку утверждения! Перечень некоторых утверждений приведен в табл. 5.1.

В соответствии с требованиями к автоматическим тестам, реализуем тесты для представленной выше программы.

Таблица 5.1

Некоторые утверждения для автоматического тестирования

Утверждение	Условие истинности утверждения
`assert( boolean, [message] )`	Истинно, если boolean
`assert_equal( expected, actual, [message] )` `refute_equal( expected, actual, [message] )`	Истинно, если expected == actual
`assert_match( pattern, string, [message] )` `refute_match( pattern, string, [message] )`	Истинно, если string =~ pattern
`assert_nil( object, [message] )` `refute_nil( object, [message] )`	Истинно, если object == nil
`assert_in_delta( expected_float, actual_float, delta, [message] )`	Истинно, если (actual_float – expected_float).abs <= delta
`assert_instance_of( class, object, [message] )`	Истинно, если object.class == class
`assert_raise( Exception,... ) {block}` `assert_nothing_raised( Exception,...) {block}`	Истинно, если блок выбрасывает или не выбрасывает исключение из списка

Пример теста сортировки по возрастанию и убыванию:

```
require './tree.rb'
# require 'test/unit' # для Ruby <= 1.9
require 'minitest/unit' # для Ruby >= 2.0

# реализуем класс теста

# class TestTree < Test::Unit::TestCase # для Ruby <= 1.9
class TestTree < Minitest::Test # для Ruby >= 2.0

  def setup # вызывается перед выполнением каждого теста
    #@items = [35, 1, 24, 2, -4, 3, 25, 4, 94, 5, 0, 6, 14, 7]
    @items = 100.times.map{ Random.rand(200) - 100 }
    @tree = Tree.new
    @items.each{ |x| @tree.insert(x) }
    @result = Array.new
    puts
  end

  def test_1 # первый тест

    @tree.inorder{|x| print "#{x} "; @result << x }
    # проверяем утверждение на равенство
```

```
    assert_equal( @items.sort, @result )
  end
  def test_2 # второй тест
    @tree.inbackorder{|x| print "#{x} "; @result << x }
    # проверяем утверждение на равенство
    assert_equal( @items.sort_by{|x| -x}, @result )
  end

  # вызывается после выполнения каждого теста. Здесь можно было не
использовать.
  def teardown
  end
end
```

Запуск теста осуществляется так же, как и запуск любой другой программы на Ruby. Итогом работы теста будет сводка о его выполнении:

```
Run options:

# Running tests:
-4 0 1 2 3 4 5 6 7 14 24 25 35 94 .
94 35 25 24 14 7 6 5 4 3 2 1 0 -4 .

Finished tests in 0.003000s, 666.6667 tests/s, 666.6667 assertions/
s.

2 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

Подробнее тему тестирования смотрите по ссылке https://en.wikibooks.org/wiki/Ruby_Programming/Unit_testing и http://docs.seattlerb.org/minitest/Minitest/Assertions.html .

Язык Ruby хорошо подходит для написания Domain Specific Language (DSL) или специализированных языков предметной области. Это способствовало тому, что именно на основе Ruby появилось множество специализированных языков, в том числе для тестирования программ, написанных на Ruby. Наиболее известный из них — RSpec (см. http://rspec.info/). В первую очередь это DSL и фреймворк для тестирования Ruby-программ. По сравнению со встроенными unit-тестами RSpec имеет развитые средства формирования отчетов о выполнении программ. Однако главное отличие заключается в том, что программа для тестирования выглядит как текст, написанный на английском языке. Приведем пример, взятый из исходной программы RSpec:

```
# Исходная программа bowling.rb
class Bowling
  def hit(pins)
end
```

```
  def score
    0
  end
end
```

Код тестируемой программы:

```
# bowling_spec.rb
require 'bowling'

describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    expect(bowling.score).to eq(0)
  end
end
```

Обратите внимание на то, что описание теста начинается со слов describe и it "returns 0 for all gutter game" do. С точки зрения языка describe и it являются методами, которые реализованы во фреймворке RSpec. Однако здесь describe определяет область тестов, а it … определяет содержимое конкретного теста. Текстовая строка после it является ничего не значащим для RSpec описанием сути теста, однако вместе они образуют вполне узнаваемое по структуре предложение. Утверждения в RSpec описываются специальными методами, которые добавляются к „expect“. Один из таких методов – to в строке expect(bowling.score).to. Его аргументом, с точки зрения синтаксиса Ruby, является метод #eq.

Контрольные вопросы и задания

1. Какие подходы для тестирования Ruby-программ существуют?
2. Что заставляет Ruby считать предъявленный текст программы именно unit-тестом?
3. Каким образом unit-тест делает заключение об успешном или не успешном прохождении теста?
4. Объясните различие подходов Minitest::Unit и RSpec.

5. Приведите примеры проверяемых утверждений для теста.

6. ОБЛАСТИ ПРИМЕНЕНИЯ ЯЗЫКА RUBY

В этом разделе дан краткий обзор тех областей, в которых может быть использован язык Ruby. Приведенные сведения не претендуют на полноту. Ввиду универсальности и высокой технологичности для написания программ язык Ruby может найти применение в очень широком диапазоне областей. Однако абсолютно универсальным языком Ruby быть не может потому, что скорость работы программ будет ниже реализации на C (если не рассматривать вызов из Ruby готовых библиотек на C), а объем оперативной памяти, требуемый при выполнении Ruby-программ, существенно выше, чем в случае C-программ.

6.1. Написание скриптов для администрирования

Поскольку Ruby в настоящее время разработан для операционных систем семейств Linux, BSD, Windows, Mac OS, этот язык можно использовать для написания служебных программ, предназначенных для контроля и управления операционной системой. Несмотря на широкое распространение скриптов на языках bash, awk для операционной системы Linux, скрипты на языке Ruby позволяют писать гораздо более понятный современным программистам код, не создавая при этом проблем с его применением. Удобство Ruby в этом случае заключается в том, что легко обеспечить работу с массивами или реализовать разбор текстовых файлов для формирования последовательности Linux-команд, а также проводить обработку результатов выполнения программ, формирующих вывод в консоль. Например, запуск консольного процесса и получение результата его выполнения может быть записан в Ruby следующими способами (список неполный):

- `result = `ls -l``
- `f = open("|ls -l")`

```
    result = f.read()
```

- `result = IO.popen(["ls", "-l"]).read.`

Первый способ широко применяется в языках bash и Perl.

Для дальнейшей обработки строки, на которую ссылается result, доступны все средства Ruby.

6.2. DSL. Возможности языка для создания новых языков

Следствием гибкости языка Ruby является возможность создания на его основе специализированных языков для конкретной предметной области – DSL (Domain Specific Language). Возможность языка игнорировать скобки при вызове методов, а также перечислять аргументы (но с некоторыми дополнительными усилиями) позволяет обеспечить возможность написания хотя и ограниченного, но связного текста на естественном языке. С использованием этой возможности были разработаны многочисленные языки DSL: RSpec, Capybara, Chef, God, Sinatra и др.

Рассмотрим более подробно создание DSL на языке Ruby. В простого примера приведём условный язык Quiz'em, который предназначен для написания тестов-опросников. Например, следующий вопрос и варианты ответа:

```
Who was the first president of the USA?
1 - Fred Flintstone
2 - Martha Washington
3 - George Washington
4 - George Jetson
Enter your answer:
```

Язык Quiz'em предназначен для описания тестов-опросников. Программа на этом языке выглядит следующим образом:

```
question 'Who was the first president of the USA?'
wrong 'Fred Flintstone'
wrong 'Martha Washington'
right 'George Washington'
wrong 'George Jetson'

question 'Who is buried in Grant\'s tomb?'
right 'U. S. Grant'
wrong 'Cary Grant'
wrong 'Hugh Grant'
wrong 'W. T. Grant'
```

Особенность Ruby здесь заключается в том, что question, right, wrong — не просто служебные слова, а полноценные методы, аргументами которых в данном случае являются строки. Пример кода для проверки возможности DSL:

```
def question(text)
  puts "Just read a question: #{text}"
end

def right(text)
  puts "Just read a correct answer: #{text}"
end

def wrong(text)
  puts "Just read an incorrect answer: #{text}"
end

load 'questions.qm'
```

В этом примере последнее действие обеспечивает загрузку файла questions.qm, содержащего вопросы и ответы в приведенном формате. В результате работы этой программы получим распечатанный текст:

```
Just read a question: Who was the first president of the USA?
Just read an incorrect answer: Fred Flintstone
Just read an incorrect answer: Martha Washington
Just read a correct answer: George Washington
Just read an incorrect answer: George Jetson
…
```

Рассмотренный пример является примером простейшего DSL, однако хорошо иллюстрирует простоту создания подобных языков на Ruby. Подробнее см. [8]. Применительно к учебному процессу может использоваться как сама возможность написания DSL для разработки его студентами, так и реализация DSL под конкретные учебные задачи.

Напомним также про особенность Ruby, заключающуюся в том, что программы реализуются в кодировке UTF-8, а идентификаторы могут быть написаны на любом языке в пределах этой кодировки. Это позволяет создавать классы, методы и переменные даже на русском языке. Приведем пример, в котором к встроенному классу Integer присоединяется метод на русском языке.

```
#coding: utf-8
class Integer
  def квадрат
    self*self
```

```
    end
end
puts 10.квадрат
puts 20.квадрат
```

В этом случае изменения ограничены лишь одним методом, однако ничто не мешает создать полноценный DSL для русскоязычных пользователей.

6.3. Тестирование программ

В случае программ, написанных на языке Ruby, очевидно, что средства для его тестирования также реализуются на Ruby. То же касается веб-приложений. Однако средства тестирования на языке Ruby применимы для так называемого браузерного тестирования, при котором браузерами с помощью библиотек типа Selenium Webdriver управляет программа тестирования. Тогда описание программы тестирования осуществляется с использованием RSpec или Capybara (см. https://github.com/teamcapybara/capybara), который является надстройкой над интерфейсом Selenium. Тестирующая программа в последнем случае будет выглядеть следующим образом:

```
describe "the signup process", :type => :feature do
  before :each do
    User.make(:email => 'user@example.com', :password => 'caplin')
  end

  it "signs me in" do
    visit '/sessions/new'
    within("#session") do
      fill_in 'Login', :with => 'user@example.com'
      fill_in 'Password', :with => 'password'
    end
    click_link 'Sign in'
    page.should have_content 'Success'
  end
end
```

Тестирование настольных приложений требует иного подхода. Существует вариант описания тестов с использованием средств операционной системы (см. Ian Dees. Scripted GUI Testing with Ruby. Pragmatic Programmers. Pragmatic Bookshelf, 2008. 192p). Это возможно, поскольку Ruby позволяет подключать любые

динамические библиотеки той операционной системы, на которой работает. Другой вариант — активно развивающаяся библиотека Sikuli
и средство для графического описания теста SikuliX (см. http://www.sikuli.org/), основной принцип которых заключается в том, чтобы построить программу тестирования в виде последовательностей нажатий на области, представленные в программе как картинки и распознаваемые на экране по фактическому изображению. Фрагмент программы тестирования на языке Ruby в IDE SikuliX приведен на рис. 6.1. Отметим, что реальный текст программы отличается от изображенного на рисунке лишь тем, что вместо рисунков подставлены имена соответствующих файлов.

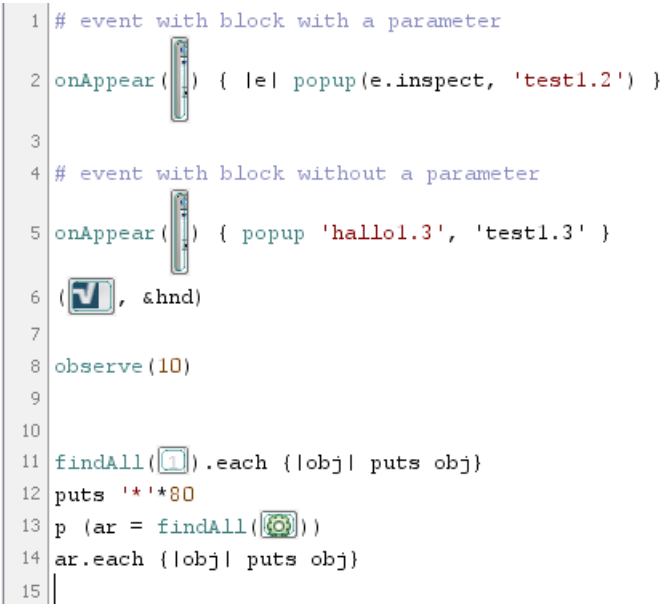

Рис. 6.1. Фрагмент программы тестирования на языке Ruby в IDE SikuliX

Несмотря на то что библиотека Sikuli реализована на языке Java с использованием реализации Ruby для Java — jRuby

возможно полноценное создание программы тестирования именно на Ruby.

6.4. Управление серверами

Для развертывания программ и массовой одновременной настройки множества серверов в Linux-системах широко используется средство под названием Puppet (см. https://puppetlabs.com/). Популярность DSL, реализуемых на Ruby, привела к созданию специального средства Chef (см. https://www.chef.io/products/chef-infra/), которое, по сути, копирует идеологию Puppet, но пользователю доступен более удобный язык для описания программ развертывания и контроля, а также все средства Ruby в полном объеме, что не доступно пользователям Puppet. Основная идея Chef заключается в том, что формируются так называемые cookbook или «поваренные книги», в которых описываются сценарии (в терминологии Chef — рецепты) установки и настройки приложения (расположение файлов, параметры конфигурации, шаблоны). Chef Server обеспечивает хранение «поваренных книг» и выдает их клиентам.

Существует большая библиотека готовых «поваренных книг» для развертывания широко используемых приложений, например PostgreSQL, Apache, Nginx. Следует отметить, что Chef широко используется именно в тех областях, где необходимо быстро настроить десятки и сотни однотипных серверов.

Другим аспектом управления серверами является запуск и отслеживание состояния выполняемых процессов. Для этого может быть использован DSL под названием God (см. http://godrb.com/).

Простейшая программа контроля выглядит следующим образом:

```
God.watch do |w|
  w.name = "simple"
  w.start = "ruby /full/path/to/simple.rb"
  w.keepalive(:memory_max => 150.megabytes,
              :cpu_max => 50.percent)
end
```

Данная программа иллюстрирует запуск процесса Ruby /full/path/to/simple.rb и отслеживание состояния его выполнения. Однако если процесс превысит потребление оперативного запоминающего устройства более 150 Мбайт или превысит 50 % загрузки процессора, он будет остановлен.

God удобен тем, что, по сути, предоставляет типовой интерфейс для написания правил контроля. В рамках данного интерфейса могут быть созданы новые правила контроля, однако код, который при этом будет написан, будет сразу же структурно понятен программисту, знакомому с God.

Применительно к учебному процессу средство Chef может использоваться как для настройки учебных классов, так и для проведения лабораторных работ по массовому конфигурированию серверов.

6.5. Создание веб-приложений

Одной из значительных задач программирования в наше время является создание веб-приложений. Следует заметить, что популярность Ruby в значительной степени обязана появлению фреймворка RubyOnRails (см. https://rubyonrails.org/), который и подчеркнул ключевые особенности языка Ruby, отсутствующие в других языках программирования. RubyOnRails представляет собой фреймворк, реализующий схему Model—View—Controller. Его особенностью является жесткая структура проекта, при которой отдельные директории отведены для контроллеров, моделей, представлений, вспомогательных модулей, а также предусмотрены тесты на все создаваемые контроллеры и модели. Для создания контроллеров и моделей имеются встроенные генераторы, которые создают необходимые файлы, код и формируют заготовки тестов с использованием Ruby unit-test или RSpec. Обращение к системе управления базами данных осуществляется с помощью специально разработанных средств объектно-реляционного преобразования (ORM). Причем реализацию ORM для RubyOnRails можно считать эталоном гибкости. Программист может вообще никогда не использовать SQL-запрос, поскольку вся работа с данными происходит на уровне Ruby-объектов.

В настоящее время этот фреймворк очень хорошо документирован, поэтому может быть рекомендован для изучения в рамках курсов, связанных с интернет-программированием.

Другим широко распространенным фрейморком является DSL Sinatra (см. http://www.sinatrarb.com/). Например, простейший код веб-сайта выглядит следующим образом:

```
require 'sinatra'

get '/hi' do
  "Hello World!"
end
```

По сути, создание веб-приложения сводится к описанию маршрутов и их обработчиков. Пример описания обработчиков для различных HTTP-запросов в форме фраз на английском языке:

```
get '/' do
    show something ..
end

post '/' do

    create something ..
end

put '/' do
    replace something ..
end
```

Дополнительно имеются средства, упрощающие написание шаблонов URL и назначение обработчиков на них. Этот фреймворк широко применяется в тех случаях, когда веб-интерфейс разрабатывается малым числом разработчиков или серверная часть веб-приложения вторична (т. е. когда основная логика выполняется на браузере). Хорошо подходит при необходимости обеспечить создание учебных веб-приложений, и при недостатке времени на изучение RubyOnRails.

6.6. Создание настольных приложений

Несмотря на полное отсутствие средств для построения графических интерфейсов пользователя в самом языке Ruby, существует большое количество библиотек, позволяющих использовать возможности библиотек wxWidgets, GTK, Qt, ncurses и др. В случае применения wxWidgets и GTK программирование сводится к манипулированию графическими примитивами так же,

как на языке C. При использовании библиотеки Qt возможно применение Qt-объектов, однако возможности, доступные C++ по переопределению классов Qt, будут недоступны.

Ruby может быть использован для написания не очень сложных графических приложений либо приложений, основной графический интерфейс которых реализован на других языках программирования. Однако в учебных целях Ruby может быть применен как для ознакомления с основными библиотеками, так и для реализации учебных задач.

6.7. Java и Ruby

Существуют реализации Ruby на языке Java. Наиболее известная из них — jRuby (см. https://www.jruby.org/). Основная особенность этой реализации по сравнению с реализацией Ruby на языке C состоит в том, что программе на языке Ruby становятся доступны все имеющиеся Java-классы. Классы на языке Ruby могут быть использованы в Java-программе. Очевидно, что полной интеграции языков достичь не удастся, однако веб-приложения на языке Ruby могут быть запущены под управлением Java-сервера приложений. Настольные приложения на Ruby могут использовать графические библиотеки Java, а для платформы Android возможно создание графических приложений с помощью средства Ruboto.

Одним из преимуществ Ruby в связке с Java является возможность использования объектной модели Ruby для классов Java. Одна из проблем Java — избыточность кода, за что этот язык часто называют церемониальным языком. jRuby позволяет изменить и это. Например, в упомянутом ранее (см. разд. 6.3) программном продукте SikuliX (см. http://sikulix.com) реализация на языке Java обработчика события «появление изображения на экране» выглядит примерно следующим образом:

```
ObserverCallBack handler = new ObserverCallBack {
        public void appeared(SikuliEvent e) {
                // делаем что-нибудь полезное
        }
  }
screen.onAppear(”picture.png”, handler)
```

Тот же обработчик на языке Ruby может быть реализован следующим образом:

```
screen.onAppear(”picture.png”) { 'делаем что-нибудь полезное' }
```

Причина столь существенного уменьшения объема кода заключается в возможности расширить Java-классы. В данном случае это реализовано в виде библиотеки примерно так (код сокращен):

```
# расширяем Java-класс из библиотеки SikuliX

class Region
  # Регистрируем универсальный класс-обработчик для всех событий
  # Класс ObserverCallBack является Java-классом
  class RObserverCallBack < ObserverCallBack
    def initialize(block)
      super()
      @block=block
    end;

   # генерируем методы-обработчкики для вызыва переданного блока
    %w(appeared vanished changed).each do |name|
      define_method(name) do |*e|
        @block.call(*e)
      end
    end
  end
  # запоминаем java-метод под другим именем
  alias_method :java_onAppear, :onAppear

  # добавляем свой метод с поддержкой блоков вместо старого
  def onAppear(target, &block)
    # вызываем старый метод и передаем ему объект обработчика,
    # который запоминает блок и вызывает его при необходимости.
    java_onAppear(target, RObserverCallBack.new(block))
  end
  # проделываем то же самое с событиями onChange, onVanish....
end
```

6.8. Бизнес-аналитика

Одним из современных направлений развития бизнес-аналитики является создание средств, позволяющих организовать сложную схему анализа и обработки данных. Среди таких проектов бизнес-аналитики следует выделить проект, предложенный на сайте https://www.knime.com/. Процесс обработки данных строится из конструктора типовых элементов, каждый из которых преобразует полученную на входе таблицу с данными в таблицу с результирующими данными. При этом могут добавляться новые колонки и изменяться таблицы целиком. Если

требуется какая-либо дополнительная обработка данных, то в рамках этого же программного пакета возможно написание тела блока на полноценном языке программирования, одним из которых является Ruby. Блоки обработки такого типа реализованы в проекте ruby4knime. В качестве примера может служить схема, приведенная на рис. 6.2.

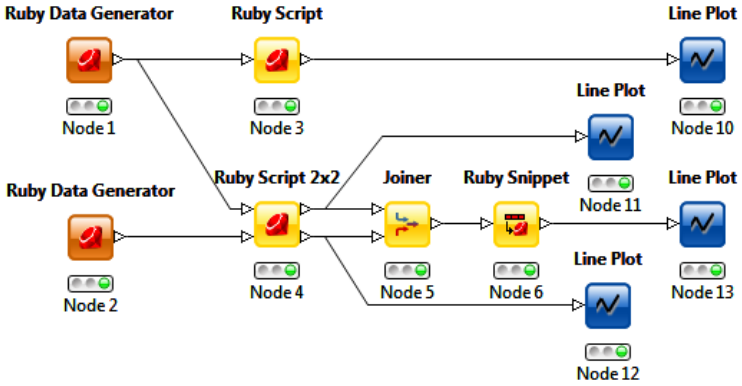

Рис. 6.2. Пример обработки данных с помощью Ruby в KNIME

Узлы, имеющие имя «Ruby...», выполняют код на языке Ruby. Например, узел Node 1 реализует генератор данных, в котором формируется таблица с тремя колонками, а данными являются результаты вычисления двух синусов. Код этого узла выглядит следующим образом:

```
0.step(Math::PI, Math::PI/1000) do |x|

  $outContainer << Cells.new.double(x).
                             double(Math.sin(x)).
                             double(Math.sin(x + Math::PI/3))
end
```

Другой пример — узел Node 6, который вычисляет разность значений разных столбцов. Его код выглядит так:

```
Cells.new.

  double(row[1].to_f).
  double(row[2].to_f - row[4].to_f)
```

6.9. Решение математических задач и визуализация

Одним из популярных инструментов в работе аналитиков является Jupyter Notebook. Этот инструмент удобен тем, что позволяет в интерактивном режиме вносить изменения в код и тут же получать ответ. Причем рабочий лист может быть использован как отчёт. Пример приводится на рис. 6.3

```
In [40]: include Math

         double_pi = PI * 2

         plot3d = Splot.new(
           'cos(x)*cos(y)',
           xrange: -double_pi..double_pi,
           yrange: -double_pi..double_pi,
           style: 'function linespoints',
           hidden3d: true,
           isosample: 30
         )
```

Out[40]:

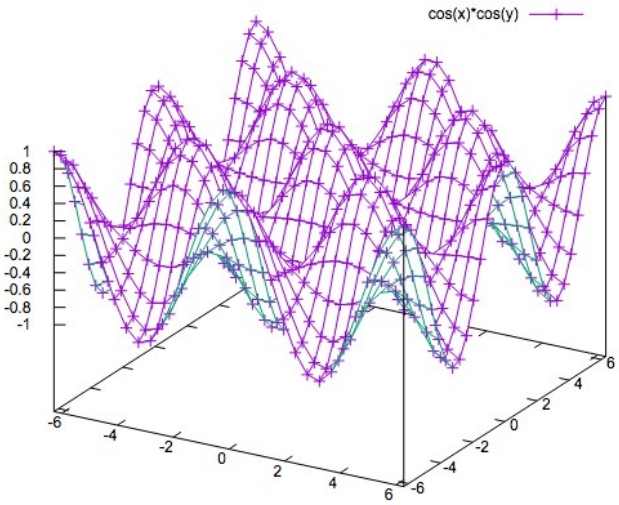

Рис. 6.3. Пример визуализации результатов Ruby в Jupiter Notebook

Перечень необходимых пакетов и шаги для установки Jupiter Notebook рассмотрены в http://sciruby.com/ и https://github.com/SciRuby/sciruby.

6.10. Прототипирование и отработка протоколов взаимодействия с периферийными устройствами

Для взаимодействия с периферийными устройствами обычно используют как низкоскоростные порты типа RS232, так и высокоскоростные типа USB. Для всех распространенных типов коммуникаций в операционной системе Linux существуют библиотеки, позволяющие реализовать взаимодействие на пользовательском уровне операционной системы. Например, взаимодействие с RS232-портами осуществляется через интерфейс файлов, взаимодействие с USB — через библиотеку libusb.

Поскольку Ruby позволяет подключать динамические библиотеки операционной системы, предусмотрены модули-переходники к ним, в частности, для USB — модуль libusb. Пример кода подключения к устройству выглядит следующим образом:

```
require "libusb"

usb    = LIBUSB::Context.new
device = usb.devices( :idVendor => 0x04b4,
                      :idProduct => 0x8613).first
device.open_interface(0) do |handle|
   handle.control_transfer( :bmRequestType => 0x40,
                            :bRequest => 0xa0,
                            :wValue => 0xe600, :wIndex => 0x0000,
                            :dataOut => 1.chr)
end
```

Возможность подключения динамических библиотек обеспечивает реализацию удобного интерфейса для описания протокола, выявления проблем и отладки процесса взаимодействия. Гибкость Ruby позволяет подготовить код для переписывания на целевых языках программирования, однако во время отладки протокола можно использовать все синтаксические средства Ruby для лаконичного описания логики взаимодействия.

Литература

Основная

1. *Фултон Хэл, Арко Андре.* Путь Ruby: 3-е издание, пер. с англ. М.: ДМК Пресс, 2016.

2. *Флэнаган Д.*, *Мацумото Ю*. Язык программирования Ruby: пер. с англ. СПб.: Питер, 2011.

3. Основной сайт Ruby. Режим доступа: https://www.ruby-lang.org/en/ (дата обращения 19.02.2020).

4. Официальная документация Ruby. Режим доступа: https://ruby-doc.org/ (дата обращения 19.02.2020).

5. Дополнительные библиотеки для Ruby. Режим доступа: https://rubygems.org/ (дата обращения 19.02.2020).

6. Учебники на русском языке. Режим доступа: https://ru.wikibooks.org/wiki/Ruby/Справочник (дата обращения 01.02.2020).

7. *Thomas D.*, *Fowler C.*, *Hunt A*. Programming Ruby 1.9 & 2.0: The Pragmatic Programmers' Guide (The Facets of Ruby) 4th Edition. Texas. Dallas: The Pragmatic Programmers, 2013., 888 pages.

8. *Russ O*. Eloquent Ruby. Pearson Education, Inc., 2011., 442 pages

Дополнительная

Набор интерактивных учебников. Режим доступа: https://rubymonk.com/ (дата обращения 19.02.2020).

Серия интерактивных учебных курсов. Режим доступа: https://www.-codecademy.com/search?query=ruby (дата обращения 19.02.2020).

Каталог библиотек и технологий Ruby. Режим доступа: https://awesome-ruby.com/ (дата обращения 19.02.2020).

УСТАНОВКА RUBY

Linux

Современные дистрибутивы Linux уже содержит Ruby. Для установки Ruby необходимо использовать встроенный менеджер пакетов zypper, yum или aptitude.

Если требуется иная версия Ruby, чем включенная в дистрибутив, то существует возможность установить средство Ruby Version Manager, которое обеспечит переключение версий по команде RVM. (Подробнее о нем см. http://rvm.io.)

Mac OS

В состав всех современных Mac OS уже включен Ruby. Если его версия недостаточна, то, так же как и в случае Linux, можно установить RVM, который позволит установить несколько версий Ruby. Для установки RVM может потребоваться менеджер пакетов Homebrew, доступный на сайте https://brew.sh/.

MS Windows

Для установки Ruby необходимо загрузить дистрибутив соответствующей версии с сайта http://rubyinstaller.org/. Основной комплект программ Ruby обычно имеет название, подобное rubyinstaller-devkit-2.7.0-1-x64.exe.

Если предполагается установка дополнительных пакетов gem, требующих компиляцию из исходных кодов по месту установки, следует загрузить по ссылке http://rubyinstaller.org/downloads/

комплект компилятора gnu C/C++ и необходимых для него библиотек DevKit. Следует выбрать файл, соответствующий версии Ruby. Например, указанному выше файлу Ruby будет соответствовать версия DevKit: DevKit-mingw64-64-4.7.2-20130224-1432-sfx.exe.

DevKit необходимо распаковать в установочную директорию Ruby, например, Ruby200\DevKit\. Перед установкой gem непосредственно в консоли следует запустить пакетный файл Ruby200\DevKit\devkitvars.bat, который присвоит соответствующие компилятору значения переменных среды.

Чтение документации

Официальная документация по языку программирования Ruby доступна на веб-сайте https://ruby-doc.org и содержит описание всех доступных классов. Однако есть некоторые особенности чтения этой документации, непонимание которых существенно затрудняет ознакомление с языком программирования Ruby. Рассмотрим пример с классом String, доступный на странице https://ruby-doc.org/core-2.7.0/String.html. См. Рис. 7.1.

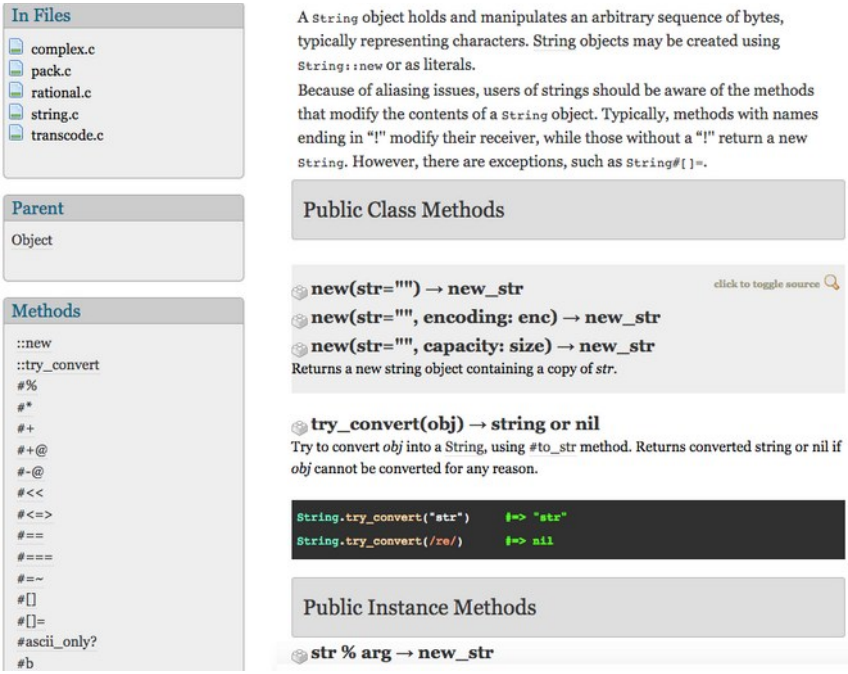

Рис. 7.1. Пример страницы документации класса String.

Левая колонка содержит сгруппированные ссылки на:

- In Files - списокк файлов, в которых реализуется этот класс. Эта информация в большей степени нужна тем, кто разрабатывает сам язык Ruby или занимается глубокой оптимизацией своих программ.
- Parents – списокк классов-предков. На этой странице видим, что у класса String есть предок Object. Такая ссылка в документации необходима потому, что приводимые ниже в колонке методы относятся только к текущему классу. То есть, методы класса предка здесь не отображаются, поэтому их надо рассматривать отдельно.
- Methods – список методов данного конкретного класса. Здесь – String.
- Included Modules – список модулей, включенных как примеси.

Следует обратить внимание на то, что часть методов обозначена с префиксом ::, в то время как другая часть – с префиком #. Это принятая в Ruby схема обозначения “public class methods” (::), то есть вызываемых от имени класса, например String.new. А также “public instance methods” (#), вызываемых от имени объекта, например “str”.sub, “1 2 3”.split.

Следующий важный вопрос – понимание примеров использования. Например для метода ::**new** видим следующее описание:

new(str="") → new_str
new(str="", encoding: enc) → new_str
new(str="", capacity: size) → new_str

Аргументы являются опциональными. Об этом говорит запись str="". Если необходимо указать кодировку или размер для резервирования памяти, необходимо использовать соответсвующие ключи :capacity или :encoding. Запись “→ new_str” говорит о том, что результатом выполнения будет создан новый объект-строка.

Следующий пример для метода #**%** (форматирование строки):

str **%** arg → new_str

В этом примере в строку с заданным форматом str подставляются аргументы arg. Результатом выполнения является новый объект-строка.

Более сложный пример с методом замены подстроки #sub:

sub(pattern, replacement) → **new_str**

sub(pattern, hash) → **new_str**

Аргументами этого метода являются шаблон pattern и вариации второго аргумента. Строка или ассоцитивный массив с именованными параметрами замены. Итогом является новый объект-строка.

sub(pattern) {|match| block } → **new_str**

Этот вариант содержит только один входной параметр, но, также, содержит блок, определяющий то, как следует преобразовывать строку. В таких случаях, обычно, документация содержит конкретные примеры. Итогом является новый объект-строка.

sub!(pattern, replacement) → **str** or **nil**

sub!(pattern) {|match| block } → **str** or **nil**

Модифицирующие методы #**sub!** Заменяют значение в исходном объекте. Обратите внимание на то, каким указано возвращаемое значение. Str (без префикса new_) в данном случае означает возврат исходной строки, для которой этот метод и был вызван. Or nil – сообщает о том, что метод может возвращать nil, если шаблон для замены не найден.

И последний пример с перечислителем **each_char**:

each_char {|cstr| block } → **str**

each_char → **an_enumerator**

В первом случае используется форма с блоком, в котором происходит какая-то обработка. При необходимости, результат может быть использован. Им является исходная строка. Во втором случае возвращается объект класса Enumerator. Его дальнейшая обработка полностью зависит от программиста.

СТИЛЬ ПРОГРАММИРОВАНИЯ И СТАТИЧЕСКАЯ ПРОВЕРКА КОДА

В отличие от языков программирования, имеющих этап компиляции, в языке Ruby изначально не предусматривались средства, проверяющие исходный код на корректность. При написании программ это приводит к невозможности быстро обнаружить опечатки или ситуации, связанные с неоднозначной интерпретацией кода. Для снижения вероятности появления подобных ошибок, а также для унификации принципов, используемых для написания кода в большом коллективе разработчиков, обычно разрабатывают различные стандарты на стили программирования. Применительно к Ruby не существует документов, имеющих статус стандартов, однако определенные рекомендации уже созданы. Пример спецификации стилей программирования Ruby:

- https://github.com/styleguide/ruby
- https://github.com/bbatsov/ruby-style-guide

Контроль соблюдения стиля программирования целесообразно проводить с помощью статических анализаторов кода. Статичность означает, что эти анализаторы не требуют реального выполнения программы. Для Ruby существует несколько статических анализаторов, имеющих различное назначение. Один из самых часто используемых — **Rubocop** (см. https://github.com/bbatsov/rubocop). Основное его назначение — именно проверка стиля и выдача рекомендаций по улучшению отдельных фрагментов кода.

Чтобы использовать rubocop, необходимо установить одноименный пакет gem, выполнив в консоли команду **gem install rubocop**.

Для проверки rb-файлов следует из консоли запустить команду

rubocop имя_файла_программы.rb

В процессе работы rubocop в консоль будут выданы предупреждения о нечитаемости кода, сообщения о возможных ошибках, а также рекомендации по улучшению кода.

Rubocop имеет режим автоматического исправления очевидных замечаний, типа некорректных отступов. Для этого его необходимо запустить с параметром командной строки **-a**.

rubocop -a имя_файла_программы.rb

Другой широко используемый анализатор — **Reek**, или «Code smell detector for Ruby» (https://github.com/troessner/reek). В отличии от Rubocop, его основное назначение — выявлять типовые ошибки, или, переводя дословно, «плохо пахнущий код». Перечень выявляемых ошибок приводится в https://github.com/troessner/reek/blob/master/docs/Code-Smells.md Среди них, использование имён переменных или методов, не несущие смысла, дубликаты кода, избыточное количество переменных и пр.

Установка Reek выполняется консольной командой **gem install reek**.

Проверка файлов запускается консольной командой в следующем формате, где options — дополнительные опции, а dir_or_source_file — перечень имён файлов или директорий, разделённых пробелами, для анализа корректности кода: **reek [options] [dir_or_source_file]***

Исходные коды программ следует проверять после каждого изменения. В промышленной разработке программ, эти проверки могут быть настроены как фильтры git-репозитория.

ОГЛАВЛЕНИЕ

Учебное издание

Самарев Роман Станиславович

Основы языка программирования Ruby

Редактор О.М. Королева
Художник Э.Ш. Мурадова
Корректор Н.В. Савельева
Компьютерная графика Т.Ю. Кутузовой
Компьютерная верстка Г.Ю. Молотковой

Оригинал-макет подготовлен
в Издательстве МГТУ им. Н.Э. Баумана.

В оформлении использованы шрифты
Студии Артемия Лебедева.

Подписано в печать 30.08.2021. Формат 70×100/16.
Усл. печ. л. 8,78. Тираж 51 экз. Изд. № 891-2020. Заказ

Издательство МГТУ им. Н.Э. Баумана.
105005, Москва, 2-я Бауманская ул., д. 5, стр. 1.
press@bmstu.ru
www.baumanpress.ru

Отпечатано в типографии МГТУ им. Н.Э. Баумана.
105005, Москва, 2-я Бауманская ул., д. 5, стр. 1.
baumanprint@gmail.com