Глава 1

Введение

Ruby — это язык динамического программирования со сложной, но выразительной грамматикой и базовой библиотекой классов с богатым и мощным API. Ruby вобрал в себя черты таких языков, как Lisp, Smalltalk и Perl, но использует грамматику, которой без особого труда смогут овладеть программисты, работающие на языках C и Java™. Ruby является абсолютным объектно-ориентированным языком, но в нем также неплохо уживаются процедурные и функциональные стили программирования. Он включает мощные потенциальные возможности для метапрограммирования, позволяющие использовать Ruby для создания языков, предназначенных для работы в конкретных предметных областях (domain-specific languages — DSL).

MATZ НА RUBY

Юкихиро Мацумото (Yukihiro Matsumoto), известный англоязычному Ruby-сообществу как Мац (Matz), является создателем Ruby и автором справочника по этому языку — «Ruby in a Nutshell» (O'Reilly) (который с обновлениями и дополнениями и превратился в эту книгу). Он говорит:

«До создания Ruby я изучил множество языков, но никогда не испытывал от них полного удовлетворения. Они были уродливее, труднее, сложнее или проще, чем я ожидал. И мне захотелось создать свой собственный язык, который смог бы удовлетворить мои программистские запросы. О целевой аудитории, для которой предназначался язык, я знал вполне достаточно, поскольку сам был ее представителем. К моему удивлению, множество программистов по всему миру испытали чувства, сходные с моими. Открывая для себя Ruby и программируя на нем, они получают удовольствие.

Разрабатывая язык Ruby, я направил всю свою энергию на ускорение и упрощение процесса программирования. Все свойства Ruby, включая объектную ориентацию, сконструированы так, чтобы при своей работе они оправдывали ожидания средних по классу программистов (например, мои собственные). Большинство программистов считают этот язык элегантным, легкодоступным и приятным для программирования».

Основная философия, заложенная Мацумото в конструкцию Ruby, сводится к его часто цитируемому высказыванию:

«Ruby предназначен для того, чтобы сделать программистов счастливыми».

1.1. Экскурсия по Ruby

Этот раздел служит несколько бессистемным путеводителем по наиболее интересным свойствам Ruby. Все, что в нем обсуждается, будет чуть позже под-

робно рассмотрено в этой книге, но этот начальный обзор даст возможность почувствовать красоту этого языка.

1.1.1. Объектная ориентированность Ruby

Начнем с того, что Ruby является полностью объектно-ориентированным языком. Каждое значение является объектом, даже простые числовые литералы и значения `true`, `false` и `nil` (`nil` — это специальное значение, свидетельствующее собственно об отсутствии какого-либо значения; это Ruby-версия `null`). Давайте применим к этим значениям метод под названием `class`. В Ruby комментарии начинаются с символа #, а стрелки вида => показывают в комментариях значения, возвращенные комментируемым кодом (это соглашение используется по всей книге):

```
1.class      # => Fixnum: число 1 относится к классу Fixnum
0.0.class    # => Float: числа с плавающей точкой относятся к классу Float
true.class   # => TrueClass: true – единственный экземпляр класса TrueClass
false.class  # => FalseClass
nil.class    # => NilClass
```

Во многих языках программирования вызовы функций и методов требуют использования круглых скобок, но ни в одном из вышеприведенных примеров кода их нет. Обычно для Ruby круглые скобки являются необязательными, и зачастую они опускаются, особенно если вызываемый метод не требует аргументов. Отсутствие круглых скобок при вызовах методов делает эти вызовы похожими на ссылки на поименованные поля или поименованные переменные объекта. Это сделано намеренно, все дело в том, что Ruby очень строг в отношении инкапсуляции своих объектов — доступ к внутреннему состоянию объекта за его пределами отсутствует. Любой подобный доступ должен иметь посредника в виде метода доступа, такого как показанный выше метод `class`.

1.1.2. Блоки и итераторы

Возможность вызова методов в отношении целых чисел — это не просто какой-то аспект Ruby, известный лишь посвященным. Им довольно часто пользуются Ruby-программисты:

```
3.times { print "Ruby! " }   # Выводит "Ruby! Ruby! Ruby! "
1.upto(9) {|x| print x }     # Выводит "123456789"
```

`times` и `upto` — это методы, выполняемые в отношении целочисленных объектов. Они представляют собой особую разновидность методов, известную как итераторы, и ведут себя как циклы. Код, помещенный в фигурные скобки, — известный как блок — связан с вызовом метода и служит в качестве тела цикла. Использование итераторов и блоков — еще одно примечательное свойство языка Ruby; хотя язык поддерживает обычный цикл `while`, большее распространение получила

реализация циклов с использованием структурных компонентов, являющихся по сути вызовами методов.

Целые числа — это не только значения, имеющие методы-итераторы. В массивах (и им подобных «перечисляемых» объектах) определен итератор по имени each, который однократно вызывает связанный с ним блок для каждого элемента массива. Каждому вызову блока передается отдельный элемент массива:

```
a = [3, 2, 1]         # Это массив литералов
a[3] = a[2] - 1       # Квадратные скобки используются для запроса
                      # и установки значений
                      # элемента массива
a.each do |elt|       # each является итератором. Блок имеет параметр elt
  print elt+1         # и выводит "4321"
end                   # Вместо скобок {} в качестве ограничителей
                      # блока использована
                      # пара do-end
```

В дополнение к each определены многие другие полезные итераторы:

```
a = [1,2,3,4]                 # Сначала задается массив
b = a.map {|x| x*x }          # Возведение элементов в квадрат: b — это [1,4,9,16]
c = a.select {|x| x%2==0 }    # Выбор четных элементов: c — это [2,4]
a.inject do |sum,x|           # Вычисление суммы всех элементов => 10
  sum + x
end
```

Хэши, подобно массивам, являются в Ruby основной структурой данных. В соответствии со своим названием, они построены на структуре данных в виде хэш-таблицы и служат для отображения произвольных объектов-ключей на объекты-значения. (Иначе говоря, хэш связывает произвольные объекты-значения с объектами-ключами.) В хэшах, как и в массивах, при работе с элементами для запроса и присваивания им значений используются квадратные скобки. При этом ожидается, что вместо целочисленных индексов в квадратных скобках будут находиться объекты-ключи. Подобно классу Array, в классе Hash так же определяется метод-итератор each. Этот метод однократно вызывает связанный с ним блок кода для каждой имеющейся в хэше пары ключ-значение и (в отличие от Array) передает блоку в качестве параметров и ключ и значение:

```
h = {                         # Хэш, отображающий названия чисел на цифры
  :one => 1,                  # "стрелки" показывают отображения: ключ=>значение
  :two => 2                   # двоеточие указывает, что это литералы обозначений
}
h[:one]                       # => 1.  Доступ к значению по ключу
h[:three] = 3                 # Добавление к хэшу новой пары ключ-значение
h.each do |key,value|         # Перебор всех пар ключ-значение
  print "#{value}:#{key}; "   # Обратите внимание, что значения подставляются
                              # в строку
end                           # Выводит "1:one; 2:two; 3:three; "
```

В качестве ключа имеющиеся в Ruby хэши могут использовать любой объект, но чаще всего используются объекты типа обозначение — Symbol. Обозначения — это неизменяемые, изолированные строки. Они могут сравниваться не по текстовому наполнению, а по идентичности (поскольку два различных объекта-обозначения никогда не будут иметь одинаковое содержимое).

Возможность связывать блок кода с вызовом метода — основное и очень мощное свойство Ruby. Хотя его применение наиболее очевидно для циклических конструкций, оно не менее полезно и для методов с однократным вызовом блока. Например:

```
File.open("data.txt") do |f|   # Открытие указанного файла
                               # и передача потока данных в блок
  line = f.readline            # Поток используется для чтения из файла
end                            # и автоматически закрывается,
                               # когда заканчивается блок

t = Thread.new do              # Этот блок запускается в новом потоке
  File.read("data.txt")        # Чтение файла в фоновом режиме
End                            # Содержимое файла доступно как значение потока
```

Отдельно следует заметить, что приведенный ранее пример использования метода Hash.each содержал следующую, довольно интересную строку кода:

```
print "#{value}:#{key}: "      # Обратите внимание, что значения подставляются
                               # в строку
```

Строка, заключенная в двойные кавычки, может включать в себя произвольные Ruby-выражения, ограниченные группой символов #{ и символом }. Значение выражения, помещенного в эти ограничители, преобразуется в строку (путем вызова метода to_s, поддерживаемого всеми объектами). Получившаяся в результате этого строка используется для замены текста выражения и его ограничителей на строковый литерал. Эта подстановка значения выражения в строку обычно называется вставкой строки.

1.1.3. Выражения и операторы Ruby

Синтаксис Ruby ориентирован на использование выражений. Такие управляющие структуры, как if, которые в других языках программирования назывались бы операторами, в Ruby представляют собой выражения. У них, как и у других, более простых выражений, есть значения, позволяющие написать следующий код:

```
minimum = if x < y then x else y end
```

Хотя все «операторы» в Ruby по своей сути являются выражениями, не все из них возвращают содержательные значения. К примеру, циклы while и определения методов являются выражениями, которые, как правило, возвращают значение nil.

Как и во многих других языках, выражения в Ruby обычно выстраиваются из значений и операторов. Большинство использующихся в Ruby знаков операций знакомы всем, кто знает Си, Java, JavaScript или подобные им языки программирования. Посмотрите на примеры самых обычных и самых необычных Ruby-операторов:

```
1 + 2                   # => 3: сложение
1 * 2                   # => 2: умножение
1 + 2 == 3              # => true: == проверка равенства
2 ** 1024               # 2 в степени 1024: в Ruby произвольная размерность
                        # целых чисел
"Ruby" + " rocks!"      # => "Ruby rocks!": объединение строк
"Ruby! " * 3            # => "Ruby! Ruby! Ruby!": повторение строки
"%d %s" % [3, "rubies"] # => "3 Rubies": Форматирование в стиле имеющегося
                        # в Python оператора printf
max = x > y ? x : y     # Условный оператор
```

Многие Ruby-операторы реализованы в виде методов, и классы могут определять (или переопределять) эти методы как угодно. (Но они не могут определять совершенно новые операторы; существует лишь фиксированный набор общепринятых операторов.) Обратите, к примеру, внимание, что знаки операций + и * для целых чисел и строк ведут себя по-разному. Но в своих собственных классах вы можете определить эти операторы как угодно. Другим подходящим примером может послужить оператор <<. Целочисленные классы `Fixnum` и `Bignum`, следуя правилам, принятым в языке программирования Си, используют этот оператор для операции поразрядного сдвига влево. В то же время (следуя C++) другие классы — строки, массивы и потоки — используют этот оператор для операции добавления. Если вы создаете новый класс, способный иметь значения, которые каким-то образом к нему добавляются, то неплохо было бы определить для него оператор <<.

Одним из самых мощных переопределяемых операторов является `[]`. Классы `Array` и `Hash` используют этот оператор для доступа к элементам массива по индексу и значениям хэша по ключу. Но в своих классах вы можете определить `[]` для чего угодно. Его можно даже определить в качестве метода, ожидающего использование нескольких аргументов, которые заключены в квадратные скобки и разделены запятыми. (Для указания подмассива или «вырезки» из массива класс `Array` воспринимает индекс и длину, заключенные в квадратные скобки.) И если нужно, чтобы квадратные скобки использовались в левой части выражения присваивания, то можно определить соответствующий оператор `[]=`. Значение в правой части присваивания будет передано в качестве конечного аргумента для метода, являющегося реализацией этого оператора.

1.1.4. Методы

Методы определяются с помощью ключевого слова `def`. Возвращаемым значением метода является то значение, которое вычисляется в его теле последним:

```
def square(x)        # Определение метода по имени square
                     # с единственным параметром x
  x*x                # Возвращение x, возведенного в квадрат
end                  # Завершение метода
```

Когда метод, подобный этому, определен за пределами класса или модуля, он фактически является глобальной функцией, а не методом, вызываемым для объекта. (Но с технической точки зрения такой метод становится закрытым методом класса `Object`.) Методы также могут быть определены для единичных объектов за счет указания перед именем метода имени объекта, для которого он определяется. Подобные методы известны как *синглтон (singleton) методы* (методы, определенные в единственном экземпляре), и именно так в Ruby определяются методы класса:

```
def Math.square(x)  # Определение метода класса для модуля Math
  x*x
end
```

Модуль `Math` является частью базовой библиотеки Ruby, и этот код добавляет к ней новый метод. Здесь проявляется ключевое свойство Ruby — классы и модули являются «открытыми» и могут быть модифицированы и расширены в процессе работы.

Параметры метода могут иметь определенные значения по умолчанию, и методы могут воспринимать произвольное количество параметров.

1.1.5. Присваивания

Имеющийся в Ruby оператор = (не подлежащий переопределению) присваивает значение переменной:

```
x = 1
```

Присваивание может сочетаться с другими операторами, такими как + и -:

```
x += 1          # Приращение x: учтите, что в Ruby нет оператора ++.
y -= 1          # Уменьшение y: также нет и оператора --.
```

Ruby поддерживает параллельные присваивания, позволяя использовать в выражении присваивания более одного значения и более одной переменной:

```
x, y = 1, 2               # То же самое, что и x = 1; y = 2
a, b = b, a               # Две переменные обмениваются значениями
x,y,z = [1,2,3]           # Значения элементов массива автоматически
                          # присваиваются переменным
```

Методам в Ruby позволено возвращать более одного значения, и в таких методах можно с пользой применить параллельное присваивание. Например:

```
# Определение метода для преобразования декартовых координат (x,y) в полярные
def polar(x,y)
  theta = Math.atan2(y,x)   # Вычисление угла
```

продолжение ⇩

```
  r = Math.hypot(x,y)        # Вычисление расстояния
  [r, theta]                 # Последнее выражение является возвращаемым значением
end

# А так мы используем этот метод с параллельным присваиванием, то
distance, angle = polar(2,2)
```

Методы, заканчивающиеся знаком равенства (=), являются специализированными, поскольку Ruby позволяет им быть вызванными с использованием синтаксиса присваивания. Если у объекта `o` есть метод по имени `x=`, то следующие две строки программного кода делают одно и то же:

```
o.x=(1)                      # Обычный синтаксис вызова метода
o.x = 1                      # Вызов метода через присваивание
```

1.1.6. Суффиксы и префиксы, состоящие из знаков пунктуации

Мы уже видели, что методы, чьи имена заканчиваются на `=`, могут быть вызваны в качестве выражения присваивания. Имена Ruby-методов могут также заканчиваться вопросительным или восклицательным знаком. Вопросительный знак используется для обозначения предикатов — методов, которые возвращают булево значение. Например, в классах `Array` и `Hash` определены методы по имени `empty?`, которые проверяют наличие в структуре данных каких-нибудь элементов. Восклицательный знак в окончании имени метода служит признаком того, что при использовании метода следует соблюдать особую осторожность. В некоторых базовых классах Ruby определены пары методов с одинаковыми именами с той лишь разницей, что у одного имени стоит в окончании восклицательный знак, а у другого — нет. Обычно метод, не имеющий восклицательного знака, возвращает модифицированную копию объекта, для которого он вызван, а метод с восклицательным знаком является методом-мутатором, который изменяет непосредственно сам объект. К примеру, у класса `Array` определены методы `sort` и `sort!`.

В дополнение к символам пунктуации в конце имен методов такие же символы будут встречаться и в начале имен переменных Ruby: глобальные переменные имеют префикс `$`, переменные экземпляров — префикс `@`, а переменные класса — префикс `@@`. Возможно, к этим префиксам нужно будет привыкать, но через некоторое время вы сможете по достоинству оценить тот факт, что префикс сообщает об области действия переменной. Префиксы нужны для устранения неоднозначностей очень гибкой грамматики Ruby. Различные префиксы могут рассматриваться в качестве платы за возможность избавиться от круглых скобок при вызовах методов.